Guide to Using DCL and Command Procedures on VAX/VMS

Order Number: AA–Y501A–TE

September 1984

This document presents key concepts and techniques for developing command procedures using the VAX/VMS DIGITAL Command Language (DCL).

Revision/Update Information: This is a new manual.

Software Version: VAX/VMS Version 4.0

digital equipment corporation
maynard, massachusetts

September 1984

The postpaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem–10	PDP	VT
DECSYSTEM–20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital

ZK–2293

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TEX, the typesetting system developed by Donald E. Knuth at Stanford University. TEX is a registered trademark of the American Mathematical Society.

Contents

FIGURES

TABLES

Preface

Intended Audience

All users of the VAX/VMS operating system can benefit from using command procedures. Command procedures can be written to perform very simple tasks and complex tasks that approximate the capabilities of a high-level language programming language.

This document presents key concepts and techniques for developing command procedures using the VAX/VMS DIGITAL Command Language (DCL). Many examples, including examples of complete command procedures, are intended to demonstrate applications of the concepts and techniques discussed. The examples included in this document include elementary, advanced, and complex command procedures.

Structure of This Document

This guide contains eight chapters and three appendixes:

- Chapter 1 is an introduction to using and developing command procedures.
- Chapter 2 describes some of the DCL concepts that you use when developing command procedures.
- Chapter 3 discusses sending input to and receiving output from command procedures.
- Chapter 4 discusses the use of symbols and lexical functions in command procedures.
- Chapter 5 discusses concepts used in designing a command procedure.
- Chapter 6 describes how to access files for input and output in a command procedure.
- Chapter 7 describes error handling within command procedures.

- Chapter 8 describes how to use command procedures in batch processes.
- Appendix A contains ten annotated sample command procedures.
- Appendix B discusses the submission of batch jobs through a card reader.
- Appendix C is a summary of lexical functions.

Associated Documents

The following documents contain information related to this guide:

- *Introduction to VAX/VMS* contains general information about using VMS.
- *VAX/VMS DCL Dictionary* contains complete descriptions of all DCL commands, defines the grammar for the DCL command language, defines the format and use of file specifications, and (of particular interest to those writing command procedures) contains many examples of specific commands.

Conventions Used in This Document

Convention	Meaning
RET	A symbol with a one- to three-character abbreviation indicates that you press a key on the terminal, for example, RET .
CTRL/x	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.

Convention	Meaning
$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.
file-spec,...	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks (″). The term apostrophe (′) is used to refer to a single quotation mark.

1 DEVELOPING COMMAND PROCEDURES

This manual describes how to write and execute command procedures. It shows how to perform common tasks such as obtaining input, writing output, establishing loops, and designing error-checking routines. This guide also describes DCL concepts that you need to understand in order to write command procedures.

A command procedure is a file that contains a list of DCL commands. When you execute a command procedure, the DCL command interpreter reads the command file and executes the commands it contains.

You can use command procedures to automate sequences of commands that you issue frequently. For example, if you always issue the DIRECTORY command after you move to a subdirectory where you keep your work files, you can write a simple command procedure to issue the SET DEFAULT and DIRECTORY commands for you. The following example, GO_DIR.COM, contains two commands:

```
$ SET DEFAULT [PERRY.ACCOUNTS]
$ DIRECTORY
```

Instead of issuing each command individually, you can execute GO_DIR.COM with the @ command:

```
$ @GO_DIR
```

This command tells the DCL command interpreter to read the file GO_DIR.COM and to execute the commands in the file. Therefore, the command interpreter sets your default directory to [PERRY.ACCOUNTS] and issues the DIRECTORY command.

You can write complex command procedures that resemble programs written in high-level programming languages. In this sense, a command procedure provides a way to write programs "written in DCL."

For example, you can revise GO_DIR.COM to allow you to move to any directory and obtain a list of the files in the directory:

```
$ INQUIRE DIR_NAME "Directory name"
$ SET DEFAULT 'DIR_NAME'
$ DIRECTORY 'DIR_NAME'
```

When you execute GO_DIR.COM, the INQUIRE command prompts for a directory name, the SET DEFAULT command moves you to that directory, and the DIRECTORY command lists the file names.

1.1 Formatting Command Procedures

Use a text editor (or the DCL command CREATE) to create and format a command procedure. When you name the command procedure, use the default file type COM. If you use this default file type, you do not have to include the file type when you execute the procedure with the @ command.

Command procedures contain DCL commands that you want the DCL command interpreter to execute and data lines that are used by these commands. Commands must begin with a dollar sign. You can start the command string immediately after the dollar sign, or you can place one or more spaces or tabs before the command string to make it easier to read.

Data lines, unlike commands, do not begin with a dollar sign. Data lines are used as input data for commands (or images). Data lines are used by the most recently issued command; these lines are not processed by the DCL command interpreter.

The following example illustrates command lines and data lines in a command procedure.

```
$ MAIL
SEND
THOMAS
MY MEMO
Do you have a few minutes to talk about the
ideas I presented in my memo?
$
$ SHOW USERS THOMAS
```

The first line is a command and must therefore start with a dollar sign. The next lines are data lines that are used by the Mail Utility; these lines must not start with dollar signs. Note that data lines must correspond to the way that the image (invoked by the command) expects the data. Therefore, the data lines provide a MAIL command (SEND), a recipient (THOMAS), a subject (MY MEMO) and the text of the mail message. When the command interpreter finds a new line that begins with a dollar sign, the Mail Utility is terminated.

Note: If you need to begin a data line with a dollar sign, use the DECK and EOD commands. See Chapter 3 for more information.

Use the following formatting conventions to make your command procedures easier to read and maintain:

- Use comments to document your command procedures.
- If a command string is very long or includes many qualifiers, continue the string on more than one line.
- Spell out command and qualifier names.
- Use labels to identify related groups of commands and use indentation to make loops and conditional coding easier to understand.

These formatting conventions are described below.

1.1.1 Documenting Command Procedures

It is good programming practice to document your command procedures so they are easy to read and to maintain. At the beginning of a command procedure, use comments to describe the procedure and any parameters that are passed to it. Also, use comments at the beginning of each block of commands to describe that section of the procedure.

The exclamation point delimits comments in command procedures. DCL considers everything to the right of an exclamation point to be a comment and ignores this information when processing the line.

The following example uses comments to document a command procedure and to make it easier to read.

```
$! This procedure sends mail to THOMAS and
$! determines whether he is currently logged
$! into the system
$!
$ MAIL
SEND
THOMAS
MY MEMO
Do you have a few minutes to talk about the
ideas I presented in my memo?
$!
$ SHOW USERS THOMAS
```

Note that whenever you insert a blank line in a command procedure, you must start the line with a dollar sign. If you omit the dollar sign, DCL interprets the line as data and issues a warning message. Also, after the dollar sign, it is a good idea to place an exclamation point. Although not required, the exclamation point helps the procedure run faster because the command interpreter ignores the line as a comment rather than searching for a command.

If you must use a literal exclamation point in a command line, enclose it in quotation marks so the command interpreter will not interpret the exclamation point as a comment delimiter. Note that you can use an exclamation point character in a data line because data lines are not processed by the command interpreter.

1.1.2 Using Multiple Lines for a Single Command

If you are writing a command that includes many qualifiers, you can make the command procedure more readable by listing the qualifiers on separate lines rather than running them together. To do this, use the hyphen as a continuation character, just as you do for interactive command continuation. Do not begin the continued line with a dollar sign. For example:

```
$  PRINT TEST.OUT -
          /AFTER=18:00 -
          /COPIES=10 -
          /QUEUE=LPB0:
```

The spaces preceding each qualifier are not required but they make the command string more readable.

1.1.3 Spelling Out Command and Qualifier Names

Even though DCL syntax rules allow truncation, you should spell out command, qualifier, and keyword names in command procedures. This helps document the command procedure, as DCL commands and qualifiers are generally named according to the functions they perform. For example, compare the following two lines:

```
$ PRINT ALPHA.LIS/COPIES=2
$ PR ALPHA/C=2
```

The first command line expresses clearly the request to print two copies of the file ALPHA.LIS. The second line is terse and may not be easily interpreted by other users (or remembered by yourself).

In addition, if you use abbreviations in command procedures you run the risk that these abbreviations may not uniquely identify commands in a future release of VAX/VMS.

1.1.4 Using Labels

In longer command procedures, you can improve readability by using labels to identify related groups of commands. When labels mark the start of a loop, use spaces or tabs to offset the commands in the body of the loop. When you indent commands, the dollar sign should still be in column 1. For example:

```
$! This procedure modifies the fifth
$! record in the file EMPLOYEES.DAT
$!
$ OPEN/READ/WRITE IN_FILE EMPLOYEES.DAT
$ COUNT = 1
$ FIND_RECORD:
$     IF COUNT .GT. 5 THEN GOTO REVISE
$     READ IN_FILE RECORD
$     COUNT = COUNT + 1
$     GOTO FIND_RECORD
$!
$ REVISE:
      .
      .
      .
```

In this example, the label FIND_RECORD delimits the beginning of the loop used to search for the fifth record in the file. The commands in the loop are indented so the body of the loop is easy to identify.

The commands IF and GOTO, and techniques for constructing loops in command procedures, are described in Chapter 5.

1.2 Executing Command Procedures

You can execute command procedures in two modes: interactive and batch. In interactive mode, the commands within a command procedure are executed as if you were typing them from your terminal. You cannot execute any other commands from your terminal until the procedure terminates. In batch mode, the system creates a separate process to run the command procedure. After you submit a batch job you can continue to use your terminal while the batch job executes.

1.2.1 Executing Command Procedures Interactively

To execute a command procedure interactively, type the @ command followed by the file specification of the procedure. If you do not enter a complete file specification, the command interpreter uses your current disk and directory and a default file type of COM. For example, the following command executes the command procedure WEATHER.COM located in your current default directory:

```
$ @WEATHER
```

When you enter the @ command, the command interpreter executes the commands in the file WEATHER.COM. Each command string in WEATHER.COM is executed sequentially and at the end of the procedure, the command interpreter issues the DCL prompt at your terminal. You can then resume interactive work.

If a command procedure is not in your default directory, or does not have the file type COM, give the complete file specification, as shown in the following example:

```
$ @DISK2:[COMMON]SETUP
```

For command procedures that you execute frequently, you can define a symbol name as a synonym for the entire command line. For example:

```
$ SETUP = "@DISK2:[COMMON]SETUP"
```

Whenever you want to run DISK2:[COMMON]SETUP.COM, you can use the symbol SETUP as if it were a command:

```
$ SETUP
```

The command interpreter replaces the symbol SETUP with its value (@DISK2:[COMMON]SETUP.COM) and then executes the procedure.

1.2.2 Submitting Command Procedures for Batch Execution

If you create and use procedures that require lengthy processing time—for example, the compilation or assembly of large source programs—submit these procedures as batch jobs so you do not tie up your terminal.

Use the SUBMIT command to execute a command procedure as a batch job. The SUBMIT command assumes your current disk and directory defaults and a default file type of COM for the command procedure. For example, to submit the command procedure WEATHER.COM (in your default directory) for batch execution, issue the command:

```
$  SUBMIT WEATHER
   Job WEATHER (queue SYS$BATCH, entry 210) started on SYS$BATCH
$
```

In this example, the system displays a message showing that the job has been queued; the message gives you the job number (210) and the name of the system queue on which the job is entered (SYS$BATCH). Then the DCL prompt is displayed and you can continue using the terminal.

Your job stays in the queue until the system is ready to run the job. Then the system creates another process for you using your account and your process characteristics. The system runs the job from that process, and deletes the process when the job is completed. For more information on batch jobs, see Chapter 8.

1.2.3 Changing Command Levels

A command level is the DCL level from which you issue commands. When you log in and type commands at your terminal, you are issuing commands from command level zero. If you execute a command procedure, the commands in the procedure are executed at command level 1. When the procedure terminates and the DCL prompt reappears on your screen, you are back at command level zero.

You can nest command procedures by using an @ command inside one procedure to execute another procedure. When you nest a command procedure, you increase the command level by one. The maximum command level you can achieve by nesting command procedures is 16.

In a batch job, command level zero is defined as the command procedure that is submitted for batch execution. If the batch job contains an @ command, then the commands in the nested procedure are executed at command level 1. In batch jobs, as in interactive mode, you can create up to 16 command levels.

1.3 Login Command Procedures

Each time you log in, the system automatically executes login command procedures. The system also executes these procedures at the beginning of every batch job you submit. The system executes two types of login command procedures:

- A system (or group) login command procedure
- Your personal login command procedure

These login procedures are described in the following sections.

1.3.1 A System or Group-Defined Login File

If a system (or group-wide) login file exists, it is executed before your personal login file. When the system login file terminates, control is passed to your personal login file. System and group login files allow your system manager to make sure that certain commands are always executed when you log in.

To establish a system or group login file, your system manager equates the logical name SYS$SYLOGIN to the appropriate login file. Your system manager can specify that this login file be used for all system users or for a certain group of users.

1.3.2 Your Personal Login File

After executing a system or group login file, the system executes your personal login file. Use your personal login file to execute commands that you want to issue every time you log in. Name your login command procedure LOGIN.COM and place it in your default login directory, unless your system manager tells you otherwise.

The following LOGIN.COM file illustrates some techniques you can use when writing your login command file.

```
$! Exit if this is a batch job or another
$! type of noninteractive process
$!
$ IF F$MODE() .NES. "INTERACTIVE" THEN EXIT     ❶
$!
$! Execute command procedures that contain
$! my symbol, logical name, and keypad definitions
$!
$ @DISK3:[MARCIA]SYMBOLS                ❷
$ @DISK3:[MARCIA]LOGICALS               ❸
$ @DISK3:[MARCIA]KEYDEF                 ❹
$!
$! Change my prompt string to an abbreviation

                                                AU$! of my node name
$!
$ NODE = F$GETSYI("NODENAME")       ❺
$ PROMPT = F$EXTRACT(0,3,NODE)
$ SET PROMPT = "''PROMPT'> "
$!
$! Type the system notices        ❻
$ TYPE SYS$SYSTEM:NOTICE.TXT
$!
$! Run a program that displays today's appointments       ❼
$ RUN DISK3:[MARCIA.PROG]REMINDER
```

❶ The F$MODE lexical function returns the mode that the process is in when the LOGIN.COM file is being executed. This statement causes the procedure to exit unless you are using the system interactively. You should test the mode at the beginning of your LOGIN.COM file to ensure that commands used only in interactive mode are not executed in any other mode; in some cases these commands can abort noninteractive processes.

❷ In this example, the symbols, logical names, and keypad definitions are kept in separate files because they are easier to maintain. Also, this keeps the LOGIN.COM file easier to read. However, if you prefer, you can keep your symbols, logical names, and keypad definitions in your LOGIN.COM file.

This command executes a command procedure that creates symbol assignments. Use symbols to create special abbreviations for commands you issue frequently. You must define global symbols or else the symbols will be deleted after the command procedure exits. The following example shows global symbol definitions in a sample SYMBOLS.COM file:

```
$ DISPLAY == "MONITOR PROCESSES/TOPCPU"
$ GO == "SET DEFAULT"
$ LP == "SHOW QUEUE/ALL SYS$PRINT"
$ SS == "SHOW SYMBOL"
$ REM == "@DISK3:[MARCIA.TOOLS]REMINDER"
$ MAIN == "SET DEFAULT DISK3:[MARCIA]"
```

❸ This command executes a command procedure that defines logical names. Use logical names to refer to files or directories that you often access. You can also use logical names to refer to usernames.

The following example shows a sample LOGICALS.COM file:

```
$! Directories
$ DEFINE HOME DISK3:[MARCIA]
$ DEFINE REV DISK3:[MARCIA.REVIEWS]
$ DEFINE TOOLS DISK3:[MARCIA.TOOLS]
$!
$! Files
$ DEFINE EQUIP DISK3:[MARCIA.LISTS]EQUIPMENT.DAT
$!
$! Users
$ DEFINE DAISY::HARRIS  JON
$ DEFINE DAISY::MOORE   JANE
```

❹ This command executes a command procedure that sets up DCL keypad definitions. You can press a defined key to type an entire command (or a command segment) with one keystroke. The following example shows keypad definitions in a sample KEYDEF.COM file:

```
$ DEFINE/KEY PF3 "SHOW USERS" /TERMINATE
$ DEFINE/KEY KP7 "SPAWN" /TERMINATE
$ DEFINE/KEY KP8 "ATTACH "
$ DEFINE/KEY KP4 "SET HOST "
```

❺ This group of commands determines the name of the node you are logging into and changes the DCL prompt to reflect the nodename. The F$GETSYI lexical function determines the nodename and the F$EXTRACT lexical function extracts the first three characters of the name. The SET PROMPT command changes the prompt from a dollar sign to the first three characters of the nodename.

❻ This command types the system notices that your system manager keeps in the file SYS$SYSTEM:NOTICE.TXT.

❼ This command runs a program that displays your daily appointments. If you have special programs that you always run after you log in, you may prefer to execute them directly from your LOGIN.COM file.

The system manager assigns the file specification for your login command procedure in the LGICMD field for your account. (Accounts are maintained in the user authorization file, or UAF.) In most installations, the login command file is called LOGIN.COM. However, if you want to execute a file other than the one named in the LGICMD field for your account, use the /COMMAND qualifier when you log in. See the description of the Login Procedure in the *VAX/VMS DCL Dictionary* for information on using the /COMMAND qualifier.

Note that your system manager can set up a captive account for you to log into, and can place the name of a special command procedure in the LGICMD field for your account. If you log into a captive account, you can perform only the functions specified in the command procedure listed in the LGICMD field of your account; you cannot use the complete set of DCL commands. For more information on captive accounts, see the *Guide to VAX/VMS System Management and Daily Operations.*

1.4 Using a Command Procedure to Define Parameters or Qualifiers

You can create a command procedure that specifies only parameters and/or qualifiers and then use the command procedure within a DCL command string. This type of command procedure is useful when there is a set of parameters or qualifiers that you frequently use with one or more particular commands. To execute the command procedure, use the @ command in the command string where you would normally use the qualifiers or parameters.

For example, suppose you frequently use the same set of qualifiers when you issue the LINK command. You could create a command procedure that contains these qualifiers, as shown below:

```
/DEBUG/SYMBOL_TABLE/MAP/FULL/CROSS_REFERENCE
```

To use this command procedure, execute it on the command line where you would otherwise place the qualifiers. For example, if you name the command procedure DEFLINK.COM, you would use the following command line to link to an object module named SYNAPSE.OBJ with the qualifiers that you specified in the command procedure:

```
$  LINK SYNAPSE@DEFLINK
```

Note that you cannot include a space before the at sign character when the command procedure contains only qualifier names.

The next example shows a command procedure named PARAM.COM that contains parameters:

```
CHAP1, CHAP2
```

To execute the procedure, use it in a command string in place of a parameter name:

```
$ DIRECTORY @PARAM
```

1.5 Testing and Debugging Command Procedures

Typically, command procedures need to be tested, then debugged. You can debug command procedures by controlling the input and output to them and by use of the commands SET VERIFY and SET NOVERIFY. When you issue the SET VERIFY command, the lines in the command procedure are displayed while the procedure is executing.

The following example shows the execution of SYMBOLS.COM when verification is set:

```
$ SET VERIFY
$ @SYMBOLS
$ DISPLAY == "MONITOR PROCESS/TOPCPU"
$ GO == "SET DEFAULT"
$ LP == "SHOW QUEUE/ALL SYS$PRINT"
$ SS == "SHOW SYMBOL"
$ REM == "@DISK3:[MARCIA.TOOLS]REMINDER"
$ MAIN == "SET DEFAULT DISK3:[MARCIA]"
$
```

Verification remains in effect until you explicitly turn it off with the SET NOVERIFY command. Note that you can also use the F$VERIFY lexical function to change verification settings. See Chapter 4 for more information on changing verification settings.

1.6 Maintaining Command Procedures

If the command procedures you develop are correctly formatted, carefully documented, and verified, their maintenance is relatively easy. However, new versions of the VAX/VMS operating system may include enhancements to the DCL command language and changes to current commands.

Generally, DIGITAL makes changes to DCL commands (and to functions of the DCL command interpreter) only to add new features, and to correct errors. However, a new release may occasionally change the format or results of a particular command, command parameter, or qualifier. For effective maintenance of command procedures, read the release notes issued with each VAX/VMS release.

2 DCL Concepts

In order to write command procedures that resemble high-level language programs, you need to use variables. A variable is a placeholder that represents a value. Use variables to represent values that can change each time a command procedure runs. To represent variables in command procedures, use logical names, symbols, or lexical functions. To manipulate variables, use DCL expressions.

This chapter gives an overview of how to use logical names, symbols, expressions, and lexical functions. This information will help you understand the examples in later chapters of this manual. If you are already familiar with these topics, skip this chapter and go to Chapter 3. For a more basic introduction, see the *Introduction to VAX/VMS*; for complete reference information, see the *VAX/VMS DCL Dictionary*.

2.1 Logical Names

A logical name is a name that can be used in place of another string. A logical name is equated to a string (called an equivalence string), or to a list of strings. When you use a logical name, the equivalence string is substituted for the logical name. The most common reason for using logical names is to represent files, directories, and devices.

For example, if the logical name TEST_FILE is equated to the equivalence string STATISTICS you can issue the following command:

```
$ RUN TEST_FILE
```

The system substitutes the equivalence string STATISTICS for the logical name TEST_FILE and runs the program STATISTICS.EXE. (The RUN command supplies the default file type EXE.)

In general, whenever a command accepts a file or a device name, the command checks whether the name you provide is a logical name. If the name is a logical name, then the system replaces the logical name with its actual value and performs the appropriate command.

When the system translates a logical name and replaces the logical name with its equivalence string, the system examines the equivalence string to see if it is also a logical name. If it is, then the system performs a second translation. This process, called iterative translation, can occur a maximum of 10 times.

2.1.1 Creating and Deleting Logical Names

You can use either the ASSIGN or DEFINE command to create a logical name; this section demonstrates the DEFINE command. The following command creates the logical name WORKFILE and equates it to the equivalence string DISK2:[WALSH.REPORTS]WORK_SUMMARY.DAT:

```
$ DEFINE WORKFILE DISK2:[WALSH.REPORTS]WORK_SUMMARY.DAT
```

To delete a logical name, use the DEASSIGN command. For example, if you no longer want to keep the logical name WORKFILE, you can delete it with the following command:

```
$ DEASSIGN WORKFILE
```

2.1.2 Using Logical Names

When you use the system interactively, you can use logical names to refer to your files and work directories. For example, if you create the logical name COMS to translate to the directory name DISK7:[WALSH.COMMAND_PROC], you can use the name COMS instead of typing the long device and directory names. The following example shows how to define the logical name COMS and use it as part of a file specification.

```
$ DEFINE COMS DISK7:[WALSH.COMMAND_PROC]
$ TYPE COMS:PAYROLL.COM
$ SET DEFAULT COMS
```

Note that when you use a logical name as part of a file specification, the logical name must be at the beginning (leftmost) part of the file specification, and must be followed by a colon. There are, however, special rules for using logical names in network file specifications. See the *VAX/VMS DCL Dictionary* for more information.

Logical names are also used to refer to the devices on your system. You should not refer to a device using its physical device name (such as DBA1); instead you should use the logical name assigned by your system manager. Also, certain commands such as ALLOCATE and MOUNT allow you to create your own logical names when you work with disks and tapes.

In command procedures, you will often use logical names when performing input and output from files. When you open a file with the OPEN command, you also create a logical name for the file. The READ, WRITE, and CLOSE commands use the logical name, not the actual file specification, to refer to the file. For example:

```
$ OPEN INFILE DISK3:[WALSH]DATA.DAT
$ READ INFILE RECORD
$ CLOSE INFILE
```

When closing the file, the CLOSE command deassigns the logical name INFILE.

2.1.3 Logical Name Tables

The system maintains logical names in logical name tables. Some logical name tables are available only to your process; others are available to other users on the system. In general, the names you work with are in the following four tables:

Process table	The names in this table are available only to your process. The actual name for your process table is LNM$PROCESS_TABLE. However, use the logical name LNM$PROCESS to refer to your process table.

Job table	The names in this table are available to your process, and to any of your subprocesses. The actual name for your job table is LNM$JOB_xxx (xxx represents a job number.) However, use the logical name LNM$JOB to refer to your job table.
Group table	The names in this table are available to all users in your group. The actual name for your group table is LNM$GROUP_xxx (xxx represents a group number.) However, use the logical name LNM$GROUP to refer to your group table.
System table	The names in this table are available to all users on the system. The actual name for your system table is LNM$SYSTEM_TABLE. However, use the logical name LNM$SYSTEM to refer to the system table.

When you use a logical name, the system looks in the process, job, group, and system tables (in that order) to obtain the name's translation. You can, however, change the way that the system translates logical names; see the *VAX/VMS DCL Dictionary* for more information.

By default, the DEFINE and DEASSIGN commands place and delete names from your process table. However, you can request a different table with the /TABLE qualifier. For example:

```
$ DEFINE/TABLE=LNM$SYSTEM REVIEWERS DISK3:[PUBLIC]REVIEWERS.DIS
```

Note that you need special privileges to place or delete names in the group or system tables; see the *VAX/VMS DCL Dictionary* for more information.

Most of the time you will create logical names in your process table. However if you are a system manager or if your work requires you to add names to shareable tables, you will need to work with the group and system tables. See the *VAX/VMS DCL Dictionary* for more information on the special privileges you need to work with shareable tables.

You can create new tables in addition to those that VAX/VMS provides. To create additional tables, use the CREATE/NAME_TABLE command.

The system maintains logical name tables in two types of directories: a process-private directory (LNM$PROCESS_DIRECTORY) and a shareable directory (LNM$SYSTEM_DIRECTORY). There is only one shareable directory table for your system, but each process has its own process-private directory. All logical name tables and any logical names that translate to tables are kept in these directory tables. For example, your table names LNM$PROCESS, LNM$JOB, and LNM$GROUP are kept in your process-private directory; LNM$SYSTEM is kept in the shareable directory.

2.1.4 Displaying Logical Names

Use the SHOW LOGICAL command to display logical names and their equivalence strings. The SHOW LOGICAL command searches certain logical name tables for the name (or names) you specify. If you do not specify a table to be searched, the SHOW LOGICAL command searches your process, job, group, and system tables. However, you can change the default search order, or you can specify other logical name tables to be searched.

The following command displays the equivalence string for the logical name INTRO.

```
$ SHOW LOGICAL INTRO
0 "INTRO" [super] = "DISK6:[OCONNELL.INTRO]" (LNM$PROCESS_TABLE)
```

This command uses the default search order and therefore searches the process, job, group, and system tables for the logical name INTRO. The number 0 indicates that INTRO is the first iteration of the translation. If the logical name has additional iterations, these will also be displayed. (Iteration numbers start with zero.)

The SHOW LOGICAL command displays, in brackets, any modes or attributes that are associated with a logical name. In the previous example, the word "[super]" indicates that the logical name was created in supervisor mode. The name in parentheses (LNM$PROCESS_TABLE) shows the table where the logical name was found.

If you do not specify a name with the SHOW LOGICAL command, then all names in the process, job, group, and system tables are displayed unless you have changed your default search order.

Note that you can use the /TABLE qualifier to specify a table for the SHOW LOGICAL command to search. For example, if you issue the command SHOW LOGICAL /TABLE=LNM$PROCESS, names in the process table are displayed.

2.1.5 Access Modes and Attributes

Logical names have access modes and attributes that are associated with the names. (The SHOW LOGICAL command displays these modes and attributes.) When you create a logical name, the name is created in supervisor mode unless you use a qualifier to specify a different mode. No attributes are applied unless you explicitly request them.

In command procedures, you usually create names in supervisor mode. However, sometimes you may want to create a temporary logical name assignment that lasts only for the execution of one program. To create a temporary assignment, use the DEFINE command with the /USER_MODE qualifier. (See Chapter 3 for more information.)

For more information on access modes and attributes, see the *VAX/VMS DCL Dictionary.*

2.1.6 Search Lists

A search list is a logical name that has more than one equivalence string. For example:

```
$ DEFINE MY_FILES DISK1:[COOPER], DISK2:[COOPER]
```

You can use a search list in any place you can use a logical name. When you use a search list, the system translates the search list using each equivalence string in the search list definition. For example:

```
$ TYPE MY_FILES:CHAP1.RNO
```

This command attempts to type the file DISK1:[COOPER]CHAP1.RNO. If this file does not exist, the system will look for the file DISK2:[COOPER]CHAP1.RNO.

For more information on how commands process search lists, see the *VAX/VMS DCL Dictionary.*

2.1.7 System-Defined Logical Names

When your system is booted (that is, when the operating system is started up), a set of system-wide logical names are created and placed in the system logical name table (LNM$SYSTEM). These logical names allow you to refer to commonly used files or devices without remembering their actual names. For example, if you want a list of your operating system's programs, you do not need to know the name of the disk and directory where these programs are stored. Instead, you can use the logical name SYS$SYSTEM as shown:

```
$ DIRECTORY SYS$SYSTEM
```

See the *VAX/VMS DCL Dictionary* for a complete list of system-wide logical name tables.

In addition to the system-wide names, every time you log in the system creates a group of logical names for your process and places these names in your process table. For example, the logical name SYS$LOGIN refers to your default device and directory when you log in. If you have changed your current defaults by using the SET DEFAULT command, you can use the following command to display a file from your initial default directory:

```
$ TYPE SYS$LOGIN:DAILY_NOTES.DAT
```

For a complete list of your default process logical names, see the *VAX/VMS DCL Dictionary*. Note that four of your default process logical names refer to process permanent files; these are described in Section 2.2.1.

2.2 Process Permanent Files

Process permanent files are files that can remain open for the life of your process. By default, DCL creates four process permanent files for you when you log in. In addition, anytime you issue a command that opens a file, DCL opens the file as a process permanent file. For example, if you open a file with the OPEN command, this file is opened as a process permanent file. The file remains open until you explicitly close the file, or until you log out.

Process permanent files are stored in a special area in memory. Note that if you keep a large number of files open at the same time, you can exhaust this area. If this occurs, close some of the files (or log out).

2.2.1 Default Process Permanent Files

By default, DCL creates and assigns logical names to four process permanent files. You often use these names in command procedures to do things such as read data from the terminal and display data. (See Chapter 3.)

Your default process permanent files are:

SYS$COMMAND	The initial file from which DCL reads input. (A file from which DCL reads input is called an input stream.) The command interpreter uses SYS$COMMAND to "remember" the original input stream.
SYS$ERROR	The default file to which DCL writes error messages generated by warnings, errors and severe errors.
SYS$INPUT	The default file from which DCL reads input.
SYS$OUTPUT	The default file to which DCL writes output. (A file to which DCL writes output is called an output stream.)

If you use the SHOW LOGICAL command to determine the equivalence string for a process permanent file, only the device portion of the string is displayed. For example:

```
$ SHOW LOGICAL SYS$INPUT
0 "SYS$INPUT" [exec] = "_TTB4:" [terminal] (LNM$PROCESS_TABLE)
```

However, the equivalence strings for process permanent files also contain special characters that the system uses to locate the correct files. These special characters are never displayed to users; they are only used internally by the system.

When you use the system interactively, SYS$INPUT, SYS$OUTPUT, and SYS$ERROR and SYS$COMMAND are all equated to your terminal. However, when you execute command procedures and submit batch jobs, DCL creates new equivalences for these logical names.

When you execute a command procedure interactively:

- SYS$INPUT is equated to the command procedure; therefore, DCL obtains commands from the command procedure. This assignment is temporary and after the command procedure terminates, SYS$INPUT has its original value.
- SYS$OUTPUT, SYS$COMMAND, and SYS$ERROR remain equated to the terminal.

When you submit a batch job:

- SYS$INPUT and SYS$COMMAND are equated to the batch job command procedure.
- SYS$OUTPUT and SYS$ERROR are equated to the batch job log file.

When you nest command procedures, the equivalence for SYS$INPUT changes to point to the command procedure that is currently executing. However, the equivalences for SYS$OUTPUT, SYS$ERROR, and SYS$COMMAND remain the same unless you explicitly change them.

2.2.2 Redefining System Logical Name Defaults

DCL provides default values for SYS$INPUT, SYS$OUTPUT, SYS$ERROR, and SYS$COMMAND that you ordinarily need not change. However, when you want to perform certain input and output operations, as described in Chapter 3, you must change these equivalences.

For information on redefining the default process permanent files, see the *VAX/VMS DCL Dictionary*.

2.3 Symbols

A symbol is a name to which you assign a character string or integer value. When you use DCL interactively, you can equate symbols to command strings and then use the symbols as command synonyms. For example, if you equate the symbol LP to the command SHOW QUEUE/ALL SYS$PRINT, you can enter:

```
$ LP
```

DCL will substitute the string SHOW QUEUE/ALL SYS$PRINT for the symbol LP.

In command procedures, symbols help you perform programming tasks. For example, you can use symbols as variables in expressions. You can also use symbols to pass parameters to and from command procedures. In addition, commands such as READ, WRITE, and INQUIRE use symbols to refer to data records. See Chapter 5 for more information on tasks you can perform using symbols.

To create a symbol, use the = (Assignment Statement). For example,

```
$ SUM = 5 + 7
$ NAME = "MORTIMER" + " SNERD"
```

The left side of the assignment statement defines the symbol name; the right side of the assignment statement contains an expression. When you create a symbol, DCL evaluates the expression and assigns the result to the symbol. If the evaluated expression is an integer, then the symbol has an integer value. If the expression evaluates to a character string, then the symbol has a string value.

In the above examples, the symbol SUM has an integer value because the expression (5 + 7) evaluates to an integer (12). The symbol NAME has a string value because the expression evaluates to a character string ("MORTIMER SNERD"). Note that you must enclose character strings in quotation marks when they are used in expressions.

To display the value of a symbol, use the SHOW SYMBOL command. For example:

```
$ SHOW SYMBOL SUM
  SUM = 12   Hex = 0000000C  Octal = 00000000014
$ SHOW SYMBOL NAME
  NAME = "MORTIMER SNERD"
```

Note that when a symbol has an integer value, the SHOW SYMBOL command displays the value in decimal, hexadecimal, and octal notation.

You can create two types of symbols, local and global. Local symbols are accessible from the current command level, and from command procedures executed from the current command level. Global symbols are accessible at all command levels. To create a local symbol, use one equal sign; to create a global symbol, use two equal signs. For example,

```
$ SS = "SHOW SYMBOL"
$ GO == "SET DEFAULT"
```

DCL places local symbols in the local symbol table for the current command level. DCL maintains a local symbol table for each command level as long as the command level is active; when a command level is no longer active, its local symbol table (and all the symbols it contains) is deleted.

DCL maintains one global symbol table for the duration of the process and places all global symbols in that table. Therefore, a global symbol exists for the duration of the process, unless the symbol is explicitly deleted.

When DCL determines the value of a symbol, it looks in the local symbol table for the current command level. If DCL does not find the symbol, it searches in the local symbol tables for previous command levels. If the symbol is not found, DCL then looks in the global symbol table.

You can delete global and local symbols with the DELETE /SYMBOL command. By default, DELETE/SYMBOL attempts to delete symbols from the current local symbol table. If you want to delete a global symbol, you must use the /GLOBAL qualifier. For example:

```
$ DELETE/SYMBOL/GLOBAL GO
```

2.4 Expressions

Expressions are values or groups of values that DCL evaluates. There are two types of expressions:

- Integer expression—an expression that evaluates to an integer
- Character string expression—an expression that evaluates to a character string

Expressions are used in symbol assignment statements (on the right side of the equal sign), in IF statements, in WRITE commands, and as arguments for lexical functions.

Expressions can contain character strings, integers, lexical functions, symbols or combinations of these values. When used in expressions, these values are called operands. If an expression contains more than one operand, the operands are connected by operators that specify the operations to be performed. The following examples illustrate expressions:

Y = 25	This integer expression contains only one operand, the number 25. Therefore, the value 25 is assigned to the symbol Y.
X = Y—8	This integer expression contains two operands: Y and 8. Y is a symbol and 8 is an integer. The minus sign is an operator indicating subtraction. When DCL evaluates this expression, it replaces the symbol Y with its actual value and performs the subtraction.

TITLE = "TWELFTH " + "NIGHT"	This string expression contains two operands: "TWELFTH " and "NIGHT". Because both operands are character strings, the plus sign indicates that the strings should be concatenated. When DCL evaluates this expression, the result is "TWELFTH NIGHT".
IF COUNT .EQ. 4 THEN EXIT	In this IF statement, DCL determines whether the integer expression COUNT .EQ. 4 is true. DCL replaces the symbol COUNT with its actual value and tests whether it equals 4. If it does, then the command procedure exits.

2.4.1 Values in Expressions

The following sections describe values that can be used in expressions. These values include:

- Character strings
- Integers
- Symbols
- Lexical functions

2.4.1.1 Character Strings

When you use a character string in an expression you must enclose it in quotation marks. (If you do not use quotation marks, DCL will think that the string is a symbol name.) To include quotation marks within a string, type two consecutive quotation marks. For example:

```
$ PROMPT = "Type ""YES"" or ""NO"""
$ SHOW SYMBOL PROMPT
  PROMPT = "Type "YES" or "NO""
```

To continue a character string over two lines, use a plus sign (for string concatenation) and a hyphen (for continuation):

```
$ ADVICE = "A STITCH IN TIME " + -
_$ "SAVES NINE"
$ SHOW SYMBOL ADVICE
  ADVICE = "A STITCH IN TIME SAVES NINE"
```

You cannot use the hyphen continuation character within a quoted character string.

2.4.1.2 Integers

Specify integers as decimal numbers (numbers in base 10), unless you use the radix operator %X (for hexadecimal) or %O (for octal). Do not place quotation marks around integers. For example:

```
$ NUM = 17
$ HEXNUM = %X1AB
```

2.4.1.3 Symbols

When you use a symbol in an expression, the symbol's value is automatically substituted for the symbol; do not surround the symbol name with quotation marks. The following example uses the symbol COUNT in an expression.

```
$ COUNT = 3
$ TOTAL = COUNT + 1
$ SHOW SYMBOL TOTAL
  TOTAL = 4    Hex = 00000004  Octal = 00000000004
```

The value for COUNT (3) is automatically substituted when the expression is evaluated and the result is assigned to the symbol TOTAL.

Remember that when you use symbols within character strings, you must use apostrophes to request symbol substitution. See Section 2.5.1 for more information.

2.4.1.4 Lexical Functions

Lexical functions return information about an item or a list of items. (These items are called arguments.) You invoke a lexical function by typing its name (which always begins with F$) and its argument list. You must always surround the argument list with parentheses.

You can use lexical functions in expressions in the same way you would normally use character strings, integers, and symbols. When you use a lexical function in an expression, DCL automatically evaluates the function and replaces the function with its return value. For example:

```
$ SUM = F$LENGTH("BUMBLEBEE") + 1
$ SHOW SYMBOL SUM
  SUM = 10   Hex = 0000000A  Octal = 00000000012
```

The F$LENGTH function returns the length of the value specified as its argument. DCL automatically determines the return value (9) and uses this value to evaluate the expression. Therefore, the result of the expression (9 + 1) is 10, and this value is assigned to the symbol SUM.

Note that each lexical function returns information as either an integer or a character string, depending on the function. Also, note that you must specify the arguments for a lexical function as either integer or character string expressions, depending on the function.

For example, the F$LENGTH function requires an argument that is a character string expression, and it returns a value that is an integer. In the previous example, the argument "BUMBLEBEE" is a character string expression and the return value (9) is an integer.

When you specify arguments for lexical functions, follow the usual rules for writing expressions: character strings are enclosed in quotation marks; integers, symbols, and lexical functions are not.

The following examples show different ways you can specify the argument for the F$LENGTH function; in each example the argument is a character string expression. The first example shows a symbol that is used as an argument:

```
$ BUG = "BUMBLEBEE"
$ LEN = F$LENGTH(BUG)
$ SHOW SYMBOL LEN
  LEN = 9    Hex = 00000009  Octal = 00000000011
```

When you use the symbol BUG as an argument, do not place quotation marks around it. The lexical function automatically substitutes the value "BUMBLEBEE" for BUG, determines the length, and returns the value 9.

This example shows an argument that contains both a symbol and a character string:

```
$ LEN = F$LENGTH(BUG + "S")
$ SHOW SYMBOL LEN
  LEN = 10    Hex = 0000000A  Octal = 00000000012
```

The symbol BUG is not enclosed in quotation marks, but the string "S" is. Before the F$LENGTH function can determine the length, the argument must be evaluated. The value represented by the symbol BUG ("BUMBLEBEE") is concatenated with the string "S"; the result is "BUMBLEBEES". The F$LENGTH function determines the length of the string "BUMBLEBEES" and returns the value 10.

The next example uses another lexical function as the argument for the F$LENGTH function. (The F$DIRECTORY function returns the name of your current default directory, including the square brackets. In this example, the current default directory is [SALMON].)

```
$ LEN = F$LENGTH(F$DIRECTORY())
$ SHOW SYMBOL LEN
  LEN = 8    Hex = 00000008  Octal = 00000000010
```

Do not place quotation marks around the F$DIRECTORY function when it is used as an argument; the function is automatically evaluated. Before the F$LENGTH function can determine the length, the result of the F$DIRECTORY must be returned. Then, the F$LENGTH determines the length of your default directory, including the square brackets.

Appendix C gives a brief description of each lexical function. For complete information, see the Lexical Functions section in Part II of the the *VAX/VMS DCL Dictionary*.

2.4.2 Operators Used in Expressions

When an expression has more than one operand, the operands are connected by operators. Table 2–5 lists the operators in the order in which they are evaluated if there are two or more operators in an expression. The operators are listed from highest to lowest precedence; that is, operators at the top of the table are performed before operators at the bottom. If an expression contains operators that have the same order of precedence, the operations are performed from left to right.

You can override the normal order of precedence by using parentheses. For example:

```
$ RESULT = 4 * (6 + 2)
$ SHOW SYMBOL RESULT
  RESULT = 32   Hex = 00000020  Octal = 00000000040
```

In this example, the parentheses force the addition to be performed before the multiplication. If you had not used the parentheses, the multiplication would have been performed first and the result would have been 26. See the *VAX/VMS DCL Dictionary* for complete information on using each operator.

Table 2–5 Summary of Operators in Expressions

Operator	Precedence	Description
+	7	Indicates a positive number
–	7	Indicates a negative number
*	6	Multiplies two numbers
/	6	Divides two numbers
+	5	Adds two numbers or concatenates two character strings
–	5	Subtracts two numbers or reduces a string
.EQS.	4	Tests if two character strings are equal

Table 2–5 (Cont.) Summary of Operators in Expressions

Operator	Precedence	Description
.GES.	4	Tests if first string is greater than or equal to second
.GTS.	4	Tests if first string is greater than second
.LES.	4	Tests if first string is less than or equal to second
.LTS.	4	Tests if first string is less than second
.NES.	4	Tests if two strings are not equal
.EQ.	4	Tests if two numbers are equal
.GE.	4	Tests if first number is greater than or equal to second
.GT.	4	Tests if first number is greater than second
.LE.	4	Tests if first number is less than or equal to second
.LT.	4	Tests if first number is less than second
.NE.	4	Tests if two numbers are not equal
.NOT.	3	Logically negates a number
.AND.	2	Combines two numbers with a logical AND
.OR.	1	Combines two numbers with a logical OR

When you write expressions that contain two or more values connected by operators, the values should all have the same data type. Values either have string or integer data types. String data includes character strings, symbols with string values, and lexical functions that return string values. Integer data includes integers, symbols with integer values, and lexical functions that return integer values.

If you use different data types, DCL converts values to the same type before evaluating the expression. In general, if you use both string and integer values, the string values are converted to integers. The only exception is when DCL performs string comparisons; in these comparisons, integers are converted to

strings. See the *VAX/VMS DCL Dictionary* for more information on how DCL converts integers to strings, and vice versa.

2.5 Symbol Substitution

In certain contexts, DCL assumes that a string of characters beginning with a letter is a symbol name or a lexical function. In these contexts, DCL will automatically try to replace the symbol or lexical function with its value. If you use a symbol or lexical function in any other context, you must use a substitution operator to request symbol substitution.

DCL automatically evaluates symbols and lexical functions when they are used in expressions. Therefore, symbols and lexical functions are automatically evaluated when they are:

- On the right side of an = or == assignment statement
- In an argument for a lexical function
- In a DEPOSIT, EXAMINE, IF, or WRITE command

In addition, symbols are automatically evaluated at the beginning of a line when the symbol is not followed by an equal sign or a colon. (The examples in the previous sections used symbols and lexical functions in places where they were automatically evaluated.)

However, you can use symbols and lexical functions in other places in command lines if you use special operators to request symbol substitution. In general, use apostrophes to request symbol substitution. However, there are special cases where you must use an ampersand instead.

2.5.1 Using Apostrophes as Substitution Operators

Use apostrophes to request symbol substitution when you use symbols and lexical functions in places where they are not automatically substituted. Place apostrophes around the symbol you are requesting substitution for. For example:

```
$ FILENAME = "CALENDAR.DAT"
$ TYPE 'FILENAME'
```

In this example, the apostrophes indicate that FILENAME is a symbol that DCL should translate. If the symbol FILENAME

is equated to the string "CALENDAR.DAT", then the file CALENDAR.DAT is displayed. If you had not included the apostrophes then DCL would not have performed the symbol substitution.

You can use apostrophes to perform symbol substitution when you concatenate two symbol names, as shown below:

```
$ NAME = "EXAMPLES"
$ TYPE = ".TST"
$ PRINT 'NAME''TYPE'
```

When the command is executed, the values for NAME and TYPE are substituted for the symbols. The two strings "EXAMPLES" and ".TST" are concatenated, forming the file-name "EXAMPLES.TST".

To request symbol substitution for a symbol within a character string, place two apostrophes before the symbol name and one apostrophe after it. For example:

```
$ PROMPT_STRING = "Creating file ''FILENAME'.TST"
```

If the current value of the symbol FILENAME is the string "BUGS", the result of the symbol substitution is the string "Creating file BUGS.TST".

2.5.2 Using Ampersands as Substitution Operators

The ampersand can also be used to request symbol substitution. The difference between the apostrophe and the ampersand is the time when substitution occurs. Symbols preceded by apostrophes are substituted during the first phase of DCL command processing; symbols preceded by ampersands are substituted during the second phase. Sometimes, the result of using these operators is the same:

```
$ TYPE 'FILENAME'
$ TYPE &FILENAME
```

If symbol FILENAME is equated to the string "BUGS", then both commands will type the file BUGS.LIS. (The TYPE command provides the default file type LIS.)

However sometimes you must use the ampersand to get the correct results. For example, when you pass parameters to a command procedure these parameters are equated to the symbols P1 through P8. You may need to use the following syntax when you are refering to parameters that were passed to a procedure:

```
$ PRINT &P'COUNT'
```

In this example, you want to print the file indicated by the first parameter, P1. First, DCL replaces the symbol COUNT with its value. If its value were 1, the resulting string would be:

```
$ PRINT &P1
```

Next, DCL performs the substitution requested by the ampersand, and substitutes the appropriate value for P1. If P1 were COMMENTS.DAT then the resulting string would be:

```
$ PRINT COMMENTS.DAT
```

In general, do not use the ampersand for symbol substitution unless it is required to correctly translate your symbols. For complete information on symbol substitution, see the *VAX/VMS DCL Dictionary*.

2.5.3 Distinguishing between Symbols and Logical Names

It is easy to confuse symbols and logical names, especially when symbols are used in place of file specifications (as shown in some previous examples). The main difference between logical names and symbols is the manner in which they are translated: DCL performs symbol substitution whereas utilities and programs perform logical name translation.

When you enter a command string, DCL examines the string to make sure it is valid. While examining the string, DCL checks for symbols in the places where it expects to find symbols, and in places indicated by substitution operators. After DCL evaluates the command string, the command starts to execute. The command must then examine the file or device specification to determine what file to process. When doing this, the command checks whether the specification contains a logical name.

3 Command Procedure I/O

This chapter discusses how to send input to and receive output from command procedures. The following topics are discussed:

- Passing input data to command procedures and to commands and images you execute from within command procedures
- Controlling output from command procedures
- Displaying information at the terminal

3.1 Passing Data to Command Procedures

When you write command procedures, you need to be able to pass input data to the procedure when it is executed. DCL provides three ways to provide this input:

- You can pass data to a command procedure using parameters
- You can issue a prompt to the user at the terminal, and the user can enter data interactively.
- You can read data from an input file.

The following sections describe how to pass parameters and how to obtain data from a user at a terminal. See Chapter 6 for information on reading data from a file.

3.1.1 Passing Parameters

When you execute a command procedure interactively, you can pass up to eight parameters by placing the values of the parameters after the file specification of the command procedure. To pass parameters to a batch job, use the /PARAMETERS qualifier when you SUBMIT the job. See Chapter 8 for more information on passing parameters to batch jobs.

Parameter values are assigned to the local symbols P1 through P8. For example:

```
$ @COPYSORT INSORT.DAT OUTSORT.DAT
```

When this command procedure is executed, two parameters are passed: INSORT.DAT and OUTSORT.DAT. The first parameter (P1) has the value INSORT.DAT; the second (P2) has the value OUTSORT.DAT.

The command procedure COPYSORT.COM can then use the symbols P1 and P2 to refer to the files it is processing. For example:

```
$ ! Copy and sort a file
$ COPY 'P1' USER_DISK:[MARY]*.*
$ SORT 'P1' 'P2'
```

This command procedure copies the file INSORT.DAT to USER_DISK:[MARY]*.*. Then, it sorts INSORT.DAT and places the sorted file in OUTSORT.DAT.

By using parameters, you can process different files each time you run the procedure, but you do not need to rewrite the procedure. Simply specify different files as parameters when you invoke the procedure.

When you list parameters on a command line, separate them with one or more spaces and/or tabs. To pass a null parameter, use a set of quotation marks as a place holder in the command string. For example:

```
@COPYFILES "" USER_DISK:[MARY]*.*
```

When COPYFILES.COM is executed, P1 will be a null string, but P2 will have the value "USER_DISK:[MARY]*.*".

You can specify a parameter as an integer, a string, or a symbol. When you specify a parameter as an integer, the integer is converted to a string when it is assigned to one of the symbols P1 through P8. For example:

```
$ @ADDER 24 25
```

In this example, the symbol P1 has the string value "24" and P2 has the string value "25". (You can, however, use the symbol P1 in both integer and character string expressions; DCL will automatically perform the necessary conversions.)

When you specify a parameter as a string, DCL automatically converts the letters in the string to uppercase. To preserve spaces, tabs, or lowercase letters, surround the string in quotation marks. If you pass fewer than eight parameters, the extra symbols are assigned null string values. If you attempt to pass more than eight parameters, you will receive an error message and the command procedure will not execute.

The following command procedure PARAMS.COM displays the parameters passed to the procedure.

```
$ ! Displaying parameters
$ SHOW SYMBOL/LOCAL/ALL
```

You can execute the procedure with the following parameters:

```
$ @PARAMS Paul Cramer
  P8 = ""
  P7 = ""
  P6 = ""
  P5 = ""
  P4 = ""
  P3 = ""
  P2 = "CRAMER"
  P1 = "PAUL"
```

P1 is assigned the value "PAUL" and P2 is assigned the value "CRAMER"; the strings are converted to uppercase because they are not surrounded in quotation marks.

The next command passes only one parameter, "Paul Cramer"; the quotation marks preserve the lowercase letters and spaces.

```
$ @PARAMS "Paul Cramer"
  P8 = ""
  P7 = ""
  P6 = ""
  P5 = ""
  P4 = ""
  P3 = ""
  P2 = ""
  P1 = "Paul Cramer"
```

When you enter a nested procedure, new symbols P1 through P8 are created in the local symbol table for the nested procedure. These symbols are assigned values passed by the invoking procedure.

3.1.2 Prompting for Input

You can use the INQUIRE command to prompt for input to be used by a command procedure. The INQUIRE command issues a prompt, reads the user's response from the terminal, and assigns it to a symbol. For example:

```
$ INQUIRE FILE "Filename"
```

This command issues the prompt "Filename: " to your terminal and assigns your response to the symbol FILE. By default, the INQUIRE command inserts a colon and a space at the end of the prompt string. You can suppress this with the /NOPUNCTUATION qualifier as described in the *VAX/VMS DCL Dictionary*.

The INQUIRE command converts your response to uppercase, replaces multiple blanks and tabs with a single space, and removes leading and trailing spaces. The INQUIRE command also performs apostrophe substitution for symbols and lexical functions. To preserve lowercase characters, multiple spaces, and tabs, enclose your response in quotation marks. Also, to include quotation marks in your response, use two sets of quotation marks in the places you want quotation marks to appear. For example:

```
$ INQUIRE FILE "Filename"
Filename: "OZ""WITCH GLENDA""::[GLENDA]SPELLS.DAT"
$ SHOW SYMBOL FILE
  FILE = "OZ"WITCH GLENDA"::[GLENDA]SPELLS.DAT"
```

Usually when you include an access control string, you enclose it in quotation marks. However, when you provide the string in response to the INQUIRE command, you must use double quotation marks.

You can also use the READ command to obtain data and retain the original case, spaces, and quotation marks. With the READ command, you do not have to enclose your response in quotation marks or use two sets of quotation marks to insert a quotation mark. For example:

```
$ READ/PROMPT="Filename: " SYS$COMMAND FILE
Filename: OZ"witch glenda"::[glenda]spells.dat
$ SHOW SYMBOL FILE
FILE = "OZ"witch glenda"::[glenda]spells.dat
```

You may find it useful to write command procedures that can either accept parameters or prompt for user input. For example,

```
$ ! Prompt for a file name if name
$ ! is not passed as a parameter
$ IF P1 .EQS. "" THEN INQUIRE P1 "Filename"
$ COPY 'P1' DISK5:[RESERVED]*.*
```

Note: When a command procedure is submitted as a batch job, the value for a symbol specified in an INQUIRE command is read from the data line following the INQUIRE command. If you do not include a data line, the symbol is assigned a null value.

3.2 Passing Data to Commands and Images

When you execute DCL commands that execute images supplied with your system or when you run your own images, you may need to provide input data. These images generally obtain input from SYS$INPUT (the default input stream) which, in a command procedure, is defined as the command procedure file. Therefore, if commands or images require input data, they will look in the command procedure file. However, if you want to provide data from another file (such as your terminal) you can redefine SYS$INPUT. The following sections describe different methods of passing data to commands and images.

3.2.1 Including Data in the Command Procedure

Use data lines to include data for commands and images within the procedure. The following command procedure runs the image CENSUS.EXE which requires input data. CENSUS.EXE obtains input from SYS$INPUT, the command procedure file.

```
$! Execute CENSUS
$ RUN CENSUS
1981
1982
1983
```

Note that the text on a data line is passed directly to CENSUS.EXE; it is not processed by DCL. You cannot include DCL symbols or expressions on data lines because DCL will not have a chance to substitute values for symbols or to evaluate expressions. Also, in a data line you cannot delimit a comment with an exclamation point (unless the image interprets the exclamation point as a delimiter).

To include data lines that begin with dollar signs in the input stream, you must define the input data in a way that prevents the command interpreter from attempting to execute the data as a command. To delimit such an input stream, you use the DECK and EOD (End of Deck) commands. For example:

```
$ ! Everything between the commands DECK and EOD
$ ! is written to the file WEATHER.COM
$ !
$ CREATE WEATHER.COM
$ DECK
$ FORTRAN WEATHER
$ LINK WEATHER
$ RUN WEATHER
$ EOD
$ !
$ ! Now execute WEATHER.COM
$ @WEATHER
$ EXIT
```

This command procedure creates and executes WEATHER.COM. Because the input lines begin with dollar signs, these lines are delimited by DECK and EOD.

You can also place programs for a compiler in the command procedure file by specifying the name of the data file as SYS$INPUT. This causes the compiler to read the program from the command procedure, rather than from another file. The following example illustrates a command procedure that

contains a FORTRAN command followed by the program's source statements:

```
$ FORTRAN/OBJECT=TESTER/LIST=TESTER SYS$INPUT
C  THIS IS A TEST PROGRAM
   A = 1
   B = 2
   STOP
   END
$ PRINT TESTER.LIS
```

In this example, the FORTRAN command uses the logical name SYS$INPUT to identify the file to be compiled. Because SYS$INPUT is equated to the command procedure, the FORTRAN compiler compiles the statements following the FORTRAN command (up to the next line that begins with a dollar sign). When the compilation is completed, two output files are created: TESTER.OBJ and TESTER.LIS. The PRINT command is then executed to print the output listing file.

Including data in the command procedure is useful when you always pass the same data to the command or image you are running. However, if you want to pass different data without rewriting the command procedure, you can redefine SYS$INPUT to either your terminal or to a data file. These techniques are described in the following sections.

3.2.2 Supplying Data to a Callable Image

When you want to run an image from a command procedure and supply data to the image interactively, redefine SYS$INPUT to be your terminal. You should make this redefinition within the command procedure using the DEFINE command. When SYS$INPUT is defined as your terminal, the image that is called from the command procedure will obtain input from your terminal rather than from data lines in the command procedure. You should use this method of obtaining input only when you execute command procedures interactively.

To redefine SYS$INPUT as your terminal, use the logical name SYS$COMMAND to indicate your terminal. (SYS$COMMAND is a logical name that is, by default, defined as your terminal when you are using the system interactively.)

When you redefine SYS$INPUT (or any other logical name for a process permanent file) it is a good idea to use the /USER_MODE qualifier. The /USER_MODE qualifier creates a temporary logical name assignment that is in effect only until the next image completes.

For example, if you want to execute the image CENSUS.EXE but you want to provide the years interactively, you can include the following commands in your procedure:

```
$ ! Execute CENSUS getting data from the terminal
$ DEFINE/USER_MODE SYS$INPUT SYS$COMMAND
$ RUN CENSUS
$ EXIT
```

The DEFINE/USER_MODE command temporarily redefines SYS$INPUT while CENSUS.EXE is running so CENSUS.EXE will obtain its input from the terminal. After CENSUS.EXE is finished, SYS$INPUT reverts to its original definition (the command procedure file.)

You will need to redefine SYS$INPUT to use, in a command procedure, any DCL command or utility that requires interactive input. For example, the next command procedure uses the EDT editor:

```
$ ! Obtain a list of your files
$ DIRECTORY
$ !
$ ! Get file name and invoke the EDT editor
$ EDIT_LOOP:
$       INQUIRE FILE "File to edit (Press RET to end)"
$       IF FILE .EQS. "" THEN EXIT
$       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND
$       EDIT 'FILE'
$       GOTO EDIT_LOOP
```

This command procedure prompts for file names until you terminate the loop with the RETURN key. When you enter a file name, the procedure automatically invokes the EDT editor to edit the file. While the editor is running, SYS$INPUT is defined as the terminal so you can enter your edits interactively.

3.2.3 Defining SYS$INPUT as a File

You can also redefine SYS$INPUT to be a data file. For example, the following command procedure executes the image CENSUS.EXE and the input data is provided in the file YEARS.DAT:

```
$ !Execute CENSUS using data in YEARS.DAT
$ DEFINE/USER_MODE SYS$INPUT YEARS.DAT
$ RUN CENSUS
$ EXIT
```

3.3 Directing Output from Command Procedures

Command procedures can direct output in the following ways. They can:

- Write data to the default output file, or can redirect data to another file
- Display error messages when certain types of errors occur
- Return values as global symbols or logical names
- Display verification of command and data lines

These methods of directing output are described in the following sections.

3.3.1 Redirecting Output from Command Procedures

In general, when you execute command procedures interactively, they display output at your terminal. This happens because command procedures send output to SYS$OUTPUT, which by default is defined as your terminal. Batch jobs send output to a batch job log file. See Chapter 8 for more information on batch job output.

You can redirect output to a file by using the /OUTPUT qualifier when you execute a command procedure. In the following example, output from SETD.COM is written to the file RESULTS.TXT instead of to the terminal:

```
$ @SETD/OUTPUT=RESULTS.TXT
```

When you issue this command, all the data that is normally displayed on your terminal is written instead to a disk file named RESULTS.TXT. To determine the outcome of the command procedure, you can use the TYPE command to display the file or the PRINT command to print it.

Note: When you use the /OUTPUT qualifier, be sure to place it immediately after the command procedure file name, with no intervening spaces. Otherwise DCL will interpret the qualifier as a parameter to be passed to the procedure.

When you redirect command procedure output to a file, the procedure will send error messages to the terminal as well as to the file that is receiving the output. (See Section 3.3.3 for more information on controlling error messages.)

3.3.2 Redirecting Output from Commands and Images

To suppress output from an individual command or image, use the /OUTPUT qualifier with that command (if the command has an /OUTPUT qualifier) or redefine SYS$OUTPUT for the duration of the command's execution.

For example, to trap the output from the DIRECTORY command, you can use the /OUTPUT qualifier. The output from the DIRECTORY command will be sent to the specified file while the output from other commands will still be sent to the terminal. For example:

```
$ ! Trap the output from DIRECTORY in NUMBER.DAT
$ ! Accumulate this information in GRAND_TOTAL.DAT
$
$ DIRECTORY/TOTAL/OUTPUT=NUMBER.DAT
$ APPEND/LOG NUMBER.DAT GRAND_TOTAL.DAT
$ EXIT
```

When you execute this procedure, do not use the /OUTPUT qualifier with the @ command because you want to suppress only the output from the DIRECTORY command.

However, some commands do not have /OUTPUT qualifiers. To trap the output from such a command, temporarily redefine SYS$OUTPUT as a file. (SYS$OUTPUT is the logical name that commands and images generally use to determine where to send output.) You can also redefine SYS$OUTPUT to trap the output from a user-written program.

The following example shows how to redirect the output from the SHOW USERS command to a file. The new definition for SYS$OUTPUT is in effect only for the execution of SHOW USERS. Then SYS$OUTPUT reverts to its default definition (the terminal).

```
$ DEFINE/USER_MODE SYS$OUTPUT SHOW_USER.DAT
$ SHOW USERS
$ !
$ ! Process the information in SHOW_USER.DAT
$ OPEN/READ INFILE SHOW_USER.DAT
$ READ INFILE RECORD
    .
    .
    .
$ CLOSE INFILE
```

To suppress output entirely from a command, temporarily define SYS$OUTPUT as a null device. For example:

```
$ DEFINE/USER_MODE SYS$OUTPUT NL:
$ APPEND NEW_DATA.DAT STATS.DAT
    .
    .
    .
```

You cannot use the DEFINE/USER_MODE command to redirect output from DCL commands that are executed within the command interpreter. However, you can redirect output from these commands by using the DEFINE command to redefine SYS$OUTPUT, and then using the DEASSIGN command to delete the definition when you are through with it. For example:

```
$ DEFINE SYS$OUTPUT TIME.DAT
$ SHOW TIME
$ DEASSIGN SYS$OUTPUT
```

After you deassign SYS$OUTPUT, its default value is restored. Table 3–1 lists the commands that are executed within the command interpreter.

Table 3–1 Commands Performed Within the Command Interpreter

=	DEPOSIT	SET PROMPT
ALLOCATE	DISCONNECT	SET PROTECTION/DEFAULT
ASSIGN	EOD	SET UIC
ATTACH	EXAMINE	SET VERIFY
CANCEL	EXIT	SHOW DEFAULT
CLOSE	GOTO	SHOW KEY
CONNECT	IF	SHOW QUOTA
CONTINUE	INQUIRE	SHOW PROTECTION
CREATE/LOGICAL_NAME_TABLE	ON	SHOW STATUS
DEALLOCATE	OPEN	SHOW SYMBOL
DEASSIGN	READ	SHOW TIME
DEBUG	RECALL	SHOW TRANSLATION
DECK	SET CONTROL	SPAWN
DEFINE	SET DEFAULT	STOP
DEFINE/KEY	SET KEY	WAIT
DELETE/KEY	SET ON	WRITE
DELETE/SYMBOL	SET OUTPUT_RATE	

3.3.3 Redirecting Error Messages

By default, command procedures send error messages to the file indicated by SYS$ERROR. You can redefine SYS$ERROR to direct error messages to a specified file. However, if you redefine SYS$ERROR to be different from SYS$OUTPUT (or if you redefine SYS$OUTPUT without also redefining SYS$ERROR), DCL commands will send error and severe error messages to both SYS$ERROR and SYS$OUTPUT. Therefore, you will receive these messages twice—once in the file indicated by the definition of SYS$ERROR, and once in the file indicated by SYS$OUTPUT. Success, informational, and warning messages will be sent only to the file indicated by SYS$OUTPUT.

If you want to suppress error messages from a DCL command, be sure that neither SYS$ERROR nor SYS$OUTPUT is equated to the terminal. For example, the following command procedure accepts a directory name as a parameter, sets the default to that directory, and purges files in the directory. In order to suppress error messages, the procedure temporarily defines SYS$ERROR and SYS$OUTPUT as the null device.

```
$ ! Purge files in a directory and suppress messages
$ SET DEFAULT 'P1'
$ ! Suppress messages
$ DEFINE/USER_MODE SYS$ERROR NL:
$ DEFINE/USER_MODE SYS$OUTPUT NL:
$ PURGE
$ EXIT
```

You can also use the SET MESSAGE command to suppress messages. For example:

```
$ ! Purge files in a directory and suppress messages
$ SET DEFAULT 'P1'
$ ! Suppress messages
$ SET MESSAGE/NOFACILITY -
             /NOIDENTIFICATION -
             /NOSEVERITY -
             /NOTEXT
$ PURGE
$ SET MESSAGE/FACILITY -
             /IDENTIFICATION -
             /SEVERITY
             /TEXT
$ EXIT
```

Note that if you run one of your own images from a command procedure and the image references SYS$ERROR, the image will send error messages only to the file indicated by SYS$ERROR—even if SYS$ERROR is different from SYS$OUTPUT. Only DCL commands and images using standard VAX/VMS error display mechanisms send error messages to both SYS$ERROR and SYS$OUTPUT when these files are different.

3.3.4 Using Global Symbols to Return Data

To return a value from a command procedure (either to a calling procedure or to DCL command level), you can assign the value to a global symbol because the global symbol can be read at any command level. You should use comments to explain the use of any global symbols.

The following command procedure, AVERAGE.COM, computes the average of up to eight numbers that are passed as parameters:

```
$ ! Find sum of the numbers (SUM) and
$ ! determine how many numbers are being averaged (COUNT)
$ IF P1 .EQS. "" THEN GOTO ERR
$ COUNT = 1
$ SUM = 0
$ LOOP:
$   IF P'COUNT' .EQS. "" THEN GOTO DONE
$   SUM = &P'COUNT' + SUM
$   COUNT = COUNT + 1
$   GOTO LOOP
$
$! Compute the average and store in the global symbol AVERAGE
$ DONE:
$   AVERAGE == SUM/(COUNT-1)
$   EXIT
$
$ ERR:
$   WRITE SYS$OUTPUT "No numbers were provided."
$   EXIT
```

You can invoke AVERAGE.COM from another procedure, and then use the global symbol AVERAGE to indicate the average. For example:

```
   .
   .
   .
$ @AVERAGE 34 60 22 86
$ WRITE SYS$OUTPUT "The average is ",AVERAGE
```

3.3.5 Using Logical Names to Return Data

You can also use logical names to return data from a nested command procedure to the calling procedure. When a logical name is defined, it is available at any command level. The following command procedure, REPORT.COM, obtains the file name for a report. Then the procedure defines the logical name REPORT_FILE, using the user-supplied name as the equivalence string. Finally, the procedure executes a program that writes a report to REPORT_FILE.

```
$! Obtain the name of a file and then run
$! REPORT.EXE to write a report to the file
$!
$ INQUIRE FILE "Name of report file"
$ DEFINE/NOLOG REPORT_FILE 'FILE'
$ RUN REPORT
$ EXIT
```

If you invoke REPORT.COM from another procedure to create the report file, the calling procedure can use the logical name REPORT_FILE to refer to the report file. For example:

```
$! Command procedure that updates data files
$! and optionally prepares reports
$ UPDATE:
   .
   .
   .
$ INQUIRE REPORT "Prepare a report [Y or N]"
$ IF REPORT THEN GOTO REPORT_SEC
$ EXIT
$!
$ REPORT_SEC:
$ @REPORT
$ WRITE SYS$OUTPUT "Report written to ", F$TRNLNM("REPORT_FILE")
$ EXIT
```

3.3.6 Verifying Command Procedure Execution

By default, only the output from commands and images is displayed when you execute command procedures interactively. If you also want to see the DCL command lines, data lines, and comment lines, use the SET VERIFY command. You can issue the SET VERIFY command either within the command procedure or at the interactive command level. If you set verification, the SET VERIFY command will be in effect for all command procedures you subsequently execute during the terminal session until you issue the SET NOVERIFY command.

For example, to display lines in a particular command procedure, you could place the SET VERIFY command at the beginning of the procedure and place the SET NOVERIFY command at the end:

```
$ ! Turn verification on
$ SET VERIFY
$ LOOP:
$     INQUIRE FILE "File name"
$     IF FILE .EQS."" THEN EXIT
$     PRINT 'FILE'
$     GOTO LOOP
$ ! Turn verification off
$ SET NOVERIFY
```

If you interactively execute a command procedure that does not contain the SET VERIFY command (and also does not contain a CTRL/Y handler) and you decide, after execution begins, that you want to see the command and data lines displayed, you can interrupt the procedure by pressing CTRL/Y as shown below:

```
$ @MASTER
CTRL/Y
Interrupt
$ SET VERIFY
$ CONTINUE
 !  The next step in this procedure is to
    .
    .
    .
```

Verifying a command procedure's execution is the principal debugging tool for detecting errors in a command procedure. If SET NOVERIFY is in effect and an error occurs, it may be difficult to determine which command caused the error. With SET VERIFY in effect, it is much easier to determine the cause of the error.

You can use the SET VERIFY command with special keywords to indicate that only command lines or data lines are to be verified. See the *VAX/VMS DCL Dictionary* for more information on SET VERIFY.

You can also use the F$VERIFY lexical function to change verification states. For example:

```
$ ! Turn verification on
$ TEMP = F$VERIFY(1)
$ LOOP:
$     INQUIRE FILE "File name"
$     IF FILE .EQS."" THEN EXIT
$     PRINT 'FILE'
$     GOTO LOOP
$ ! Turn verification off
$ TEMP = F$VERIFY(0)
```

See Chapter 4 for information on how to save verification settings before you change them.

3.4 Writing Data to the Terminal

There are several ways that you can write data to the terminal. The most common ways are with the WRITE command and the TYPE command, as described in the following sections.

3.4.1 Using the WRITE Command

In command procedures that are executed interactively, you can use the WRITE command to display lines at the terminal. To do this, specify the output file as SYS$OUTPUT. You must express the data you want to write as a string expression; the data can be a character string, a symbol, a lexical function, or a combination of these entities.

The following example shows different ways you can use the WRITE command:

```
$ ! writing a character string
$ WRITE SYS$OUTPUT "Two files were written."
$ !
$ ! writing a symbol
$ FILE = "STAT1.DAT"
$ WRITE SYS$OUTPUT FILE
$ !
$ ! writing a character string that contains a symbol
$ AFILE = "STAT1.DAT"
$ BFILE = "STAT2.DAT"
$ WRITE SYS$OUTPUT "''AFILE' and ''BFILE' were written."
$ !
$ ! writing a list of items
$ WRITE SYS$OUTPUT AFILE, "and ",BFILE," were written."
$ EXIT
```

To display a long line, continue the line over two lines. For example:

```
$ ! writing a long line
$ WRITE SYS$OUTPUT "REPORT BY MARY JONES" + -
" PREPARED APRIL 15, 1984"
$ EXIT
```

To include quotation marks in a string you are writing, use two sets of quotation marks in the places you want quotes to appear. For example:

```
$ ! including quotes
$ WRITE SYS$OUTPUT "SUMMARY OF ""Q AND A"" SESSION"
$ EXIT
```

3.4.2 Using the TYPE Command

To display text that is several lines long and does not require symbol substitution, use the TYPE command. The TYPE command writes data from the file you specify to SYS$OUTPUT. For interactive command procedures, if you specify SYS$INPUT as the file you are typing from, the TYPE command will read data from the data lines that follow and display these lines on the terminal. For example:

```
$ ! Using TYPE to display lines
$ TYPE SYS$INPUT
REPORT BY MARY JONES
PREPARED APRIL 15, 1984
SUBJECT: Analysis of Tax Deductions for 1983
$ EXIT
```

Note that when you use the TYPE command to display data, the lines are not processed by DCL. Therefore, symbols and expressions are not evaluated. If you need to evaluate symbols or expressions, use the WRITE command to display data.

3.4.3 Displaying Files

To display a file, use the TYPE command. The following statement displays the file STAT1.DAT on the terminal:

```
$ TYPE STAT1.DAT
```

4 Using Symbols and Lexical Functions

This chapter shows the types of information you can obtain and manipulate using symbols and lexical functions. It includes how to get information about:

- Your process
- The system
- Files and devices
- Logical names
- Strings
- Data types

This chapter gives general information for each lexical function; for complete descriptions, see the *VAX/VMS DCL Dictionary*. In addition, Appendix C lists each lexical function and its arguments.

Many lexical functions return information that you can also get from DCL commands. However, in a command procedure you can manipulate information more easily if you obtain it from a lexical function, rather than from a command. For example, you can use either the F$ENVIRONMENT function or the SHOW DEFAULT command to obtain the name of your current default directory. If you use the F$ENVIRONMENT function, you can assign the result to a symbol, and then use this symbol later in the procedure, as shown below:

```
$ DIR_NAME = F$ENVIRONMENT("DEFAULT")
$ SET DEFAULT DISK4:[TEST]
   .
   .
   .
$ SET DEFAULT 'DIR_NAME'
```

The F$ENVIRONMENT function returns the current default disk and directory, and stores this value in the symbol DIR_NAME. At the end of the procedure, you use the symbol DIR_NAME to restore the default with the SET DEFAULT command.

If you had used the SHOW DEFAULT command, you could not have assigned this output to a symbol. Instead, you would have needed to redirect the output to a file. To reset the default directory, you would have needed to open the file containing the SHOW DEFAULT output, read and parse the record containing the directory name, reset the directory, and close the file. Then you would have needed to delete the file containing the SHOW DEFAULT output.

4.1 Obtaining Information About Your Process

You can use the following lexical functions to obtain information about your process:

F$DIRECTORY	Returns the current default directory string.
F$ENVIRONMENT	Returns information about the command environment for your process.
F$GETJPI	Returns accounting, status, and identification information about your process, or about other processes on the system.
F$MODE	Shows the mode in which your process is executing.
F$PRIVILEGE	Indicates whether your process has the specified privileges.
F$PROCESS	Returns the name of your process.
F$SETPRV	Sets the specified privileges. This function also indicates whether the specified privileges were previously enabled before you used the F$SETPRV function.
F$USER	Returns your user identification code.
F$VERIFY	Indicates whether verification is on or off.

You often change process characteristics for the duration of a command procedure, and then restore them. Table 4–1 shows process characteristics that are commonly changed in command procedures; it also gives the lexical functions that save these characteristics and the DCL commands that change and restore the original settings.

Note that if you save process characteristics, you may need to ensure that an error or CTRL/Y interrupt does not cause the procedure to exit before restoring the original characteristics. See Chapter 7 for more information on handling errors and CTRL/Y interrupts.

Table 4–1 Commonly Changed Process Characteristics

Characteristic	Operation	Command/Lexical Function
Control characters	To save:	F$ENVIRONMENT("CONTROL")
	To restore:	SET CONTROL
DCL prompt	To save:	F$ENVIRONMENT("PROMPT")
	To restore:	SET PROMPT
Default protection	To save:	F$ENVIRONMENT("PROTECTION")
	To restore:	SET PROTECTION/DEFAULT
Key state	To save:	F$ENVIRONMENT("KEY_STATE")
	To restore:	SET KEY
Message format	To save:	F$ENVIRONMENT("MESSAGE")
	To restore:	SET MESSAGE
Privileges	To save:	F$PRIVILEGE or F$SETPRV
	To restore:	F$SETPRV or SET PROCESS /PRIVILEGES
Verification	To save:	F$VERIFY or F$ENVIRONMENT
	To restore:	F$VERIFY or SET VERIFY

4.1.1 Changing Verification Settings

One common technique in command procedures is to turn verification off for the duration of a procedure. This prevents users from displaying a procedure's contents while executing

the procedure. In command procedures, you can set or disable two types of verification:

Procedure verification	Displays each command line as it is being executed
Image verification	Displays each data line as it is being processed

The SET [NO]VERIFY command and the F$VERIFY function turn both types of verification on or off unless you explicitly request that only one type of verification be changed.

In general, you keep your verification settings the same. That is, you will keep them both on or off. However, you do not have to keep them the same. Therefore, before you change these settings in a command procedure, you should save each verification setting separately. For example:

```
$ ! Save each verification state
$ ! Turn both states off
$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(0)
    .
    .
    .
$ $ ! Restore original verification states
$ SAVE_VERIFY_IMAGE = F$VERIFY(SAVE_VERIFY_PROCEDURE,-
      SAVE_VERIFY_IMAGE)
```

The F$ENVIRONMENT function returns the current image verification setting, and assigns this value to the symbol SAVE_VERIFY_IMAGE. Next, the F$VERIFY function returns the current procedure verification setting and assigns this value to the symbol SAVE_VERIFY_PROCEDURE. The F$VERIFY function also turns off both image and procedure verification.

Note that you could have used the F$ENVIRONMENT function to obtain the procedure verification setting, and then you could have used the F$VERIFY function to turn verification off. However, it is shorter to use F$VERIFY to accomplish both tasks in one command line.

At the end of the procedure, you use the F$VERIFY function to restore the original settings (specified by the symbols SAVE_VERIFY_PROCEDURE and SAVE_VERIFY_IMAGE.) The value returned in SAVE_VERIFY_IMAGE is not used by the procedure.

4.1.2 Changing Default Protection

Another characteristic that is often changed is the default file protection code. The following command procedure changes the default protection associated with files created while the procedure is executing. The original default is restored before the procedure terminates.

```
$ SAVE_PROT = F$ENVIRONMENT("PROTECTION")
$ SET PROTECTION = (SYSTEM:RWED, OWNER:RWED, GROUP, WORLD)
   .
   .
   .
$ SET PROTECTION='SAVE_PROT'/DEFAULT
$ EXIT
```

Note that the F$ENVIRONMENT function returns the default protection code, and the return value uses the syntax required by the SET PROTECTION command. This allows you to use the symbol SAVE_PROT with the SET PROTECTION command.

4.2 Obtaining Information About the System

You can use the following lexical functions to obtain information about the system:

F$GETSYI	Returns information about your local system or about a node in your cluster (if your system is part of a cluster.)
F$IDENTIFIER	Converts identifiers from named to numeric format, and vice versa.
F$MESSAGE	Returns the message text associated with a status code.
F$PID	Returns the process identification number for processes that you are allowed to examine.
F$TIME	Returns the current date and time.

The following sections describe some commonly used system information.

4.2.1 Determining Your Node Name

If your system is part of a network or a cluster where you can log into many different nodes, you can set the DCL prompt to indicate which node you are currently using. To do this, include the F$GETSYI function in your login command procedure to determine the node name. Then use the SET PROMPT command to set a unique prompt for the node. For example:

```
$ NODE = F$GETSYI("NODENAME")
$ SET PROMPT = "''NODE'$ "
    .
    .
    .
```

If you want to use only a portion of the node name in your prompt string, use the F$EXTRACT function to extract the appropriate characters. (See Section 4.5.2 for more information on extracting characters.)

4.2.2 Obtaining Information About Processes

You can use the F$PID function to get the process identification number (PID) for all processes that you are allowed to examine. You can obtain PIDs for all processes on the system if you have WORLD privilege; you can obtain PIDs for all processes in your group if you have GROUP privilege. If you have neither GROUP nor WORLD privilege, you can obtain only the PID for your process.

The following example shows how to obtain and display the PIDs for the processes you are allowed to examine. At the beginning of the procedure, the time is displayed:

```
$ WRITE SYS$OUTPUT F$TIME()
$!
$ CONTEXT = ""
$ START:
$ ! Obtain and display PIDs until
$ ! F$PID returns a null string
$ !
$ PID = F$PID(CONTEXT)
$ IF PID .EQS. "" THEN EXIT
$ WRITE SYS$OUTPUT "Pid --- ''PID'"
$ GOTO START
```

In this example, the system uses the symbol CONTEXT to hold a pointer into the system list of PIDs. Each time through the loop, the system changes the pointer to locate the next PID in the list. The procedure exits after all PIDs have been displayed.

After you obtain a PID, you can use the F$GETJPI function to obtain specific information about the process. For example, you can enhance the above procedure so that it displays the PID and the UIC for each process:

```
$ CONTEXT = ""
$ START:
$ ! Obtain and display PIDs and UICs
$ !
$ PID = F$PID(CONTEXT)
$ IF PID .EQS. "" THEN EXIT
$ UIC = F$GETJPI(PID,"UIC")
$ WRITE SYS$OUTPUT "Pid --- ''PID'   Uic--- ''UIC' "
$ GOTO START
```

Note that you can shorten this command procedure by including the F$GETJPI function within the WRITE command, as shown below:

```
$ CONTEXT = ""
$ START:
$ PID = F$PID(CONTEXT)
$ IF PID .EQS. "" THEN EXIT
$ WRITE SYS$OUTPUT "Pid --- ''PID'   Uic --- ''F$GETJPI(PID,"UIC")'"
$ GOTO START
```

4.3 Obtaining Information About Files and Devices

You can use the following lexical functions to obtain information about files and devices:

F$FILE_ATTRIBUTES	Returns information about file attributes.
F$GETDVI	Returns information about a specified device.
F$PARSE	Parses a file specification, and returns the requested field(s).
F$SEARCH	Searches a directory for a file.

The following sections describe some commonly used file information.

4.3.1 Searching for a File in a Directory

Before processing a file, a command procedure may need to test whether the file currently exists. To do this, use the F$SEARCH function. For example, the following command procedure uses F$PARSE to apply a device and directory string to the file STATS.DAT. Then the procedure uses the F$SEARCH function to determine whether STATS.DAT is present in DISK3:[JONES.WORK]. If it is, the command procedure processes the file. Otherwise, the command procedure prompts for another input file.

```
$ FILE = F$PARSE("STATS.DAT","DISK3:[JONES.WORK]",,,"SYNTAX_ONLY")
$ IF F$SEARCH(FILE) .EQS. "" THEN GOTO GET_FILE
$ PROCESS_FILE:
    .
    .
    .
$ GET_FILE:
$   INQUIRE FILE "File name"
$   GOTO PROCESS_FILE
```

After determining that a file exists, the procedure can use the F$PARSE or the F$FILE_ATTRIBUTES function to get additional information about the file. For example:

```
$ IF F$SEARCH("STATS.DAT") .EQS. "" THEN GOTO GET_FILE
$ PROCESS_FILE:
$     NAME = F$PARSE("STATS.DAT", "NAME")
    .
    .
    .
$ GET_FILE:
$   INQUIRE FILE "File name"
$   GOTO PROCESS_FILE
```

4.3.2 Deleting Old Versions of Files

If a command procedure creates files that you do not need after the procedure terminates, delete or purge these files before you exit from the procedure. Use the PURGE command to delete all copies except the most recent one; use the DELETE command to delete all copies.

To avoid error messages when you delete files from a command procedure, use the F$SEARCH function to verify that a file exists before you try to delete it. For example, you can write a command procedure that creates a file TEMP.DAT only if

certain modules are executed. The following line issues the DELETE command only if TEMP.DAT exists:

```
$ IF F$SEARCH("TEMP.DAT") .NES. "" THEN DELETE TEMP.DAT;*
```

4.4 Translating Logical Names

You can use the following lexical functions to translate logical names:

F$LOGICAL	Returns the equivalence string for a logical name.
F$TRNLNM	Returns either the equivalence string or the requested attributes for a logical name.

Note: The F$TRNLNM function supersedes the F$LOGICAL function that was used in earlier versions of VAX/VMS. You should use F$TRNLNM (rather than F$LOGICAL) to ensure that your command procedure processes logical names using the current system techniques.

In some situations, you may want to use logical names rather than symbols as variables in command procedures. For example, programs can access logical names more easily than they can access DCL symbols. Therefore, to pass information to a program that you run from a command procedure, obtain the information using a symbol. Then use the DEFINE command to equate the value of the symbol to a logical name.

The following example tests whether the logical name NAMES has been defined. If it has, the procedures runs PAYROLL.EXE. Otherwise, the procedure obtains a value for the symbol FILE and uses this value to create the logical name NAMES. For example:

```
$ ! Make sure that NAMES is defined
$ IF F$TRNLNM("NAMES") .NES. "" THEN GOTO ALL_SET
$ INQUIRE FILE "File with employee names"
$ DEFINE NAMES 'FILE'
$ !
$ ! Run PAYROLL, using the file indicated by NAMES
$ ALL_SET:
$ RUN PAYROLL
    .
    .
    .
```

This example obtains input with the INQUIRE command and defines this input as the logical name NAMES. Next, the procedure executes the program PAYROLL.EXE. This program uses the logical name NAMES to refer to the file of employee names.

You can also use the F$TRNLNM function to assign the value of a logical name to a symbol. For example:

```
$ DEFINE NAMES DISK4:[JONES]EMPLOYEE_NAMES.DAT
$ RUN PAYROLL
    .
    .
    .
$ FILE = F$TRNLNM(NAMES)
$ WRITE SYS$OUTPUT "Finished processing ",FILE
```

This command procedure defines a logical name that is used in the program PAYROLL. At the end of the procedure, the WRITE command displays a message indicating that the file was processed. Because the WRITE command does not translate logical names, you need to equate the logical name (NAMES) to a symbol (FILE). Then you can use the symbol FILE to display the file name.

4.5 Manipulating Strings

You can use the following lexical functions to manipulate character strings:

F$CVTIME	Returns information about a time string.
F$EDIT	Edits a character string.
F$ELEMENT	Extracts an element from a string in which the elements are separated by delimiters.
F$EXTRACT	Extracts a section of a character string.
F$FAO	Formats an output string.
F$LENGTH	Determines the length of a string.
F$LOCATE	Locates a character or a substring within a string, and returns the offset.

4.5.1 Determining if a String or Character is Present

One common reason for examining strings is to determine whether a character (or substring) is present within a character string. To do this, use the F$LENGTH and the F$LOCATE

functions. If the value returned by the F$LOCATE function equals the value returned by the F$LENGTH function, then the character you are looking for is not present.

The following procedure requires a file name that includes the version number. To determine whether a version number is present, the procedure tests whether a semicolon (which indicates a version number) is included in the file name that the user enters:

```
$ INQUIRE FILE "Enter file (include version number)"
$ IF F$LOCATE(";", FILE) .EQ. F$LENGTH(FILE) THEN -
     GOTO NO_VERSION
   .
   .
   .
```

The F$LOCATE function returns the offset for the semicolon. Offsets start with 0; thus, if the semicolon were the first character in the string, the F$LOCATE function would return the integer 0. If the semi-colon is not present within the string, the F$LOCATE function returns an offset that is one more than the offset of the last character in the string. This value is the same as the length returned by F$LENGTH.

4.5.2 Extracting Part of a String

To extract a portion of a string, use either the F$EXTRACT function or the F$ELEMENT function. Use the F$EXTRACT function to extract a substring that starts at a defined offset; use the F$ELEMENT function to extract part of a string between two delimiters. In order to use either of these functions, you must know the general format of the string you are parsing.

Note: You do not need to use F$EXTRACT or F$ELEMENT to parse file specifications or time strings. Instead, use F$PARSE or F$CVTIME to extract the desired portion of these strings.

The following command procedure uses the F$EXTRACT function to extract the group portion of UIC. This allows the procedure to execute a different set of commands depending on the user's UIC group.

```
$ UIC = F$USER()
$ GROUP_LEN = F$LOCATE(",",UIC) - 1
$ GROUP = F$EXTRACT(1,GROUP_LEN, UIC)
$ GOTO 'GROUP'_SECTION
    .
    .
    .
$ WRITERS_SECTION
    .
    .
    .
$ MANAGERS_SECTION
    .
    .
    .
```

First, the procedure determines the UIC with the F$USER function. Next, the procedure determines the length of the group name by using F$LOCATE to locate the offset of the comma. The comma separates the group from the user portion of a UIC. Everything between the left bracket and the comma is part of the group name. For example, the group name from the UIC [WRITERS,SMITH] is WRITERS.

After determining the length, the procedure extracts the name of the group with the F$EXTRACT function. The name starts with offset 1, and ends with the character before the comma. Finally, the procedure directs execution to the appropriate label.

Note that you can determine the length of the group name at the same time you extract it. For example:

```
$ UIC = F$USER()
$ GROUP = F$EXTRACT(1, F$LOCATE(",",UIC) - 1, UIC)
$ GOTO 'GROUP'_SECTION
```

If a string contains a delimiter that separates different parts of the string, use the F$ELEMENT function to extract the part that you want. For example, in a protection code, each type of access is separated by a comma. For example:

```
$ PROT = F$ENVIRONMENT("PROTECTION")
$ SHOW SYMBOL PROT
PROT = "SYSTEM=RWED, OWNER=RWED, GROUP=RE, WORLD"
```

Thus, you can use F$ELEMENT to obtain different types of access by extracting the portions of the string between the commas. To determine SYSTEM access, obtain the first element; to determine OWNER access, obtain the second element, and so on.

The following example extracts the world access portion (the fourth element) from your default protection code. Note that when you use the F$ELEMENT function, element numbers start with zero. Therefore use the integer 3 to specify the fourth element. For example:

```
$ PROT = F$ENVIRONMENT("PROTECTION")
$ WORLD_PROT = F$ELEMENT(3,",",PROT)
    .
    .
    .
```

In this example, the F$ELEMENT function returns everything between the third comma and the end of the string. Thus, if your default protection allowed read access for world users, the string ("WORLD=R") would be returned.

After you obtain the world access string, you may need to examine it further. For example:

```
$ PROT = F$ENVIRONMENT("PROTECTION")
$ WORLD_PROT = F$ELEMENT(3,",",PROT)
$ IF F$LOCATE("=", WORLD_PROT) .EQ. F$LENGTH(WORLD_PROT) -
  THEN GOTO NO_WORLD_ACCESS
    .
    .
    .
```

4.5.3 Formatting Output Strings

You can use the WRITE command to write a string to a record. For example, the following command procedure uses the WRITE command to display the process name and process identification number for processes on the system.

```
$ ! Initialize context symbol to get PIDs
$ CONTEXT = ""
$ ! Write headings
$ WRITE SYS$OUTPUT "Process Name     PID"
$ !
$ GET_PID:
$ PID = F$PID(CONTEXT)
$ IF PID .EQS. "" THEN EXIT
$ WRITE SYS$OUTPUT F$GETJPI(PID,"PRCNAM"),"     ", F$GETJPI(PID,"PID")
$ GOTO GET_PID
```

Note that the output from the WRITE command inserts five spaces between the process name and and the username, but the columns do not line up. For example:

```
Process Name     PID
MARCHESAND      2CA0049C
TRACTMEN       2CA0043A
FALLON       2CA0043C
ODONNELL       2CA00453
PERRIN       2CA004DE
CHAMPIONS       2CA004E3
```

To line up the columns, you can use the F$FAO function to define record fields and place the process name and username in these fields. When you use the F$FAO function, use a control string to define the fields in the record; then you specify the values to be placed in these fields. For example, following procedure uses the F$FAO function to define a 16 character field and a 12 character field. The F$FAO function places the process name in the first field, skips a space, and then places the PID in the second field.

```
$ ! Initialize context symbol to get PIDs
$ CONTEXT = ""
$ ! Write headings
$ WRITE SYS$OUTPUT "Process Name     PID"
$ !
$ GET_PID:
$ PID = F$PID(CONTEXT)
$ IF PID .EQS. "" THEN EXIT
$ LINE = F$FAO("!16AS !12AS", F$GETJPI(PID,"PRCNAM"), F$GETJPI(PID,"PID"))
$ WRITE SYS$OUTPUT LINE
$ GOTO GET_PID
```

Now when you execute the procedure, the columns will be correctly formatted:

```
Process Name     PID
MARCHESAND       2CA0049C
TRACTMEN         2CA0043A
FALLON           2CA0043C
ODONNELL         2CA00453
PERRIN           2CA004DE
CHAMPIONS        2CA004E3
```

Another way to format fields in a record is to use a character string overlay. The following example uses an overlay to place the process name in the first 16 characters (starting at offset 0) of the symbol RECORD. Then the PID is placed in the next 12 characters (starting at offset 17):

```
$ ! Initialize context symbol to get PIDs
$ CONTEXT = ""
$ ! Write headings
$ WRITE SYS$OUTPUT "Process Name     PID"
$ !
$ GET_PID:
$ PID = F$PID(CONTEXT)
$ IF PID .EQS. "" THEN EXIT
$ RECORD[0,16]:= 'F$GETJPI(PID,"PRCNAM")'
$ RECORD[17,12]:= 'F$GETJPI(PID,"PID")'
$ WRITE SYS$OUTPUT RECORD
$ GOTO GET_PID
```

This procedure produces the same type of formatted columns you created with the F$FAO function:

```
Process Name     PID
MARCHESAND       2CA0049C
TRACTMEN         2CA0043A
FALLON           2CA0043C
ODONNELL         2CA00453
PERRIN           2CA004DE
CHAMPIONS        2CA004E3
```

Note, however, that the F$FAO function is more powerful than a character string overlay; you can perform a wider range of output operations with the F$FAO function.

4.6 Manipulating Data Types

You can use the following lexical functions to convert data from strings to integers, and vice versa:

F$CVSI	Extracts bit fields from a character string and converts the result, as a signed value, to an integer.
F$CVUI	Extracts bit fields from a character string and converts the result, as an unsigned value, to an integer.
F$INTEGER	Converts a string expression to an integer.
F$STRING	Converts an integer expression to a string.
F$TYPE	Determines the data type of a symbol.

4.6.1 Converting Data Types

Use the F$INTEGER and F$STRING functions to convert between integers and strings. For example, the following command procedure converts data types. If you enter a string, the command procedure shows the integer equivalent. If you enter an integer, the command procedure shows the string equivalent. Note how the F$TYPE function is used to form a label name in the GOTO statement; F$TYPE returns "STRING" or "INTEGER" depending on the data type of the symbol.

```
$ IF P1 .EQS. "" THEN INQUIRE P1 "Value to be converted"
$ GOTO CONVERT_'F$TYPE(P1)'
$
$ CONVERT_STRING:
$ WRITE SYS$OUTPUT "The string ''P1' is converted to ''F$INTEGER(P1)'"
$ EXIT
$
$ CONVERT_INTEGER:
$ WRITE SYS$OUTPUT "The integer ''P1' is converted to ''F$STRING(P1)'"
$ EXIT
```

4.6.2 Evaluating Expressions

Some commands, such as INQUIRE and READ, accept string data only. If you use these commands to obtain data that you want to evaluate as an integer expression, use the F$INTEGER function to convert and evaluate this data. For example:

```
$ INQUIRE EXP "Enter integer expression"
$ RES = F$INTEGER('EXP')
$ WRITE "Result is",RES
```

The following example shows sample output from this command procedure:

```
Enter integer expression: 9 + 7
Result is 16
```

Note that you must place apostrophes around the symbol EXP when you use it as an argument for the F$INTEGER function. This causes DCL to substitute the value for EXP during the first phase of symbol substitution. In the above example, the value "9 + 7" is substituted. When the F$INTEGER function processes the argument "9 + 7", it evaluates the expression and returns the correct result.

4.6.3 Determining Whether a Symbol Exists

Use the F$TYPE function to determine whether a symbol exists; the F$TYPE function returns a null string if a symbol is undefined. For example:

```
   .
   .
   .
$ IF F$TYPE(TEMP) .EQS. "" THEN TEMP = "YES"
$ IF TEMP .EQS. "YES" THEN GOTO TEMP_SEC
   .
   .
   .
```

This procedure tests whether the symbol TEMP has been previously defined. If it has, then the current value of TEMP is retained. If TEMP is not defined, then the IF statement assigns the value "YES" to TEMP.

5 Design and Logic

The normal flow of execution in a command procedure is sequential: the commands in the procedure are executed, in order, until the end of the file is reached. However, in many cases you will want to control (1) whether certain statements are executed or (2) the conditions under which the procedure should continue execution.

This chapter discusses the commands that you can use in a command procedure to control or alter the flow of execution:

- The IF command tests the value of an expression and executes a given command string based on the result of the test.
- The GOTO command transfers control to a labeled line in the procedure.
- The Execute Procedure command invokes (or calls) another command procedure and creates another command level.
- The EXIT and STOP commands terminate the current procedure and restore control either to the calling command procedure or to command level 0, respectively.

5.1 Design

Before writing a command procedure, analyze the tasks you want to perform. You may find it helpful to perform the tasks interactively to identify the steps you need to perform the tasks. In particular, look for:

Variables	Data that change each time you perform the task
Conditionals	Conditions that must be tested each time you perform the task
Iterations	Groups of commands that are performed repetitively until a condition is met

For example, you decide to write a command procedure to

perform the commands required to clean up a directory. Before you start, you clean up one of your directories interactively and notice that you:

- Issue the DIRECTORY command to see what files are in the directory
- Issue the PURGE command to get rid of back versions of files
- Issue the DELETE command to delete files you no longer want to keep
- Issue the TYPE command to remember what a file contains before you decide whether to delete it
- Issue the PRINT command to print a hard copy of a file before you delete it

After determining the commands you use to clean a directory, you identify the following variables: the command to be executed and the file to be processed. These variables change each time you perform the procedure.

Next, you identify the following conditionals. First, you need to determine which command to perform. If the requested command is not one of the commands that the procedure knows how to perform, you need to issue an error message and get another command. Also, you need to determine when to terminate the procedure.

Finally, you decide that the command procedure will repeat the following sequence of commands, until the directory is clean: obtain a command, see if you're done, perform the command if the command is valid, and obtain another command.

5.2 Coding

To make the command procedure easier to understand and to maintain, try to write the statements so that the procedure executes straight through from the first command to the last command. This section shows the steps you go through to code a command procedure to clean up a directory. This example command procedure used in this section is called CLEAN.COM.

5.2.1 Obtaining Variables

First, decide how to obtain the values for your variables. You can use any of the methods for obtaining input described in Chapter 3. However, one of the most common methods for obtaining values for variables is to use the INQUIRE command to equate the values to symbols. In this command procedure, you obtain the command to execute from the user who is executing the procedure:

```
$    INQUIRE COMMAND -
     "Enter command (DELETE, DIRECTORY, PRINT, PURGE, TYPE)"
```

5.2.2 Coding the General Design

According to your design, the first thing you want to do after you obtain a command is to see whether you are through, or whether you need to perform the command. To do this, you decide to add an EXIT command so you can test whether the user entered "EXIT". For example:

```
$    INQUIRE COMMAND -
     "Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
$    IF COMMAND .EQS. "EXIT" THEN EXIT
```

Next, you need to determine which command to execute. To do this, check each possible command against the command issued by the user. If you find a match, you will execute the command. If you don't, go on to the next command. If there is no match after you have checked for each valid command, you issue an error message.

To test whether a condition is true, use the IF...THEN commands. To change the flow of execution, use the GOTO command to direct the flow of execution to a label in the command procedure. Also, use program "stubs" as placeholders for the code that performs each command. A stub is a temporary section of code that you use in your procedure while you test the design. After you get the overall design to work correctly, you can go back and replace each stub with the correct code. For example:

```
$    INQUIRE COMMAND -
     "Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
$    IF COMMAND .EQS. "EXIT" THEN EXIT
$!
$!Execute if user entered DELETE
$ DELETE:
$       IF COMMAND .NES. "DELETE" THEN GOTO DIRECTORY
$       WRITE SYS$OUTPUT "This is the DELETE section."
$!
$!Execute if user entered DIRECTORY
$ DIRECTORY:
$       IF COMMAND .NES. "DIRECTORY" THEN GOTO PRINT
$       WRITE SYS$OUTPUT "This is the DIRECTORY section."
$!
$!Execute if user entered PRINT
$ PRINT:
$       IF COMMAND .NES. "PRINT" THEN GOTO PURGE
$       WRITE SYS$OUTPUT "This is the PRINT section."
$!
$!Execute if user entered PURGE
$ PURGE:
$       IF COMMAND .NES. "PURGE" THEN GOTO TYPE
$       WRITE SYS$OUTPUT "This is the PURGE section."
$!
$!Execute if user entered TYPE
$   TYPE:
$       IF COMMAND .NES. "TYPE" THEN GOTO ERROR
$       WRITE SYS$OUTPUT "This is the TYPE section."
$!
$ ERROR:
$       WRITE SYS$OUTPUT "You entered an invalid command."
$
$ EXIT
```

Now that you have the lines that test your conditionals, finish the design by adding the loop that allows you to obtain a command, process the command, and repeat the process until the user issues the EXIT command. To do this, direct the flow of execution back to the beginning of the procedure after the procedure executes a command. Leave the loop only when the user enters the EXIT command. For example:

```
$ GET_COMMAND_LOOP:
$    INQUIRE COMMAND -
     "Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
$    IF COMMAND .EQS. "EXIT" THEN GOTO END_LOOP
$!
$!Execute if user entered DELETE
$    DELETE:
$       IF COMMAND .NES. "DELETE" THEN GOTO DIRECTORY
$       WRITE SYS$OUTPUT "This is the DELETE section."
$       GOTO GET_COMMAND_LOOP
$!
$!Execute if user entered DIRECTORY
$    DIRECTORY:
$       IF COMMAND .NES. "DIRECTORY" THEN GOTO PRINT
$       WRITE SYS$OUTPUT "This is the DIRECTORY section."
$       GOTO GET_COMMAND_LOOP
$!
$!Execute if user entered PRINT
$   PRINT:
$       IF COMMAND .NES. "PRINT" THEN GOTO PURGE
$       WRITE SYS$OUTPUT "This is the PRINT section."
$       GOTO GET_COMMAND_LOOP
$!
$!Execute if user entered PURGE
$   PURGE:
$       IF COMMAND .NES. "PURGE" THEN GOTO TYPE
$       WRITE SYS$OUTPUT  "This is the PURGE section."
$       GOTO GET_COMMAND_LOOP
$!
$!Execute if user entered TYPE
$   TYPE:
$       IF COMMAND .NES. "TYPE" THEN GOTO ERROR
$       WRITE SYS$OUTPUT "This is the TYPE section."
$       GOTO GET_COMMAND_LOOP
$!
$   ERROR:
$       WRITE SYS$OUTPUT "You entered an invalid command."
$       GOTO GET_COMMAND_LOOP
$!
$ END_LOOP:
$ WRITE SYS$OUTPUT "Directory ''F$DIRECTORY()' has been cleaned."
$ EXIT
```

5.2.3 Testing and Debugging

Once you have written the code using program stubs, you can test the overall logic of the command procedure. Test all possible paths of execution. For example to test CLEAN.COM, issue each command, try giving an invalid command, and then exit using the EXIT command. For example:

```
$ @CLEAN
Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE): DELETE
This is the DELETE section.
    .
    .
    .
Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE): EXIT
```

Use the SET VERIFY and SHOW SYMBOL commands to help debug command procedures. When verification is set, you can see errors and the lines that generate them. For example, the following section of CLEAN.COM contains a spelling error. You can determine where the error occurs by turning verification on before you execute the procedure.

```
$ SET VERIFY
$ @CLEAN
$ GET_COMMAND_LOOP:
$     INQUIRE COMMAND -
      "Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE):  EXIT
$     IF COMMAND .EQS. "EXIT" THEN GOTO END_LOP
%DCL-W-USGOTO, target of GOTO not found - check spelling and presence of label
```

The label END_LOP is spelled incorrectly. To correct the error, change the label to END_LOOP.

The next example uses the SHOW SYMBOL command to determine how the symbol COMMAND is defined.

```
$ SET VERIFY
$ @CLEAN
$ GET_COMMAND_LOOP:
$     INQUIRE COMMAND -
      "ENTER COMMAND (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
ENTER COMMAND (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE):  EXIT
$ SHOW SYMBOL COMMAND
 COMMAND = "EXIT"
$     IF COMMAND .EQS. "exit" THEN GOTO END_LOOP
    .
    .
    .
```

In this example, the SHOW SYMBOL command shows that the symbol COMMAND has the value "EXIT". (The INQUIRE command automatically converts input to uppercase.) However, the IF statement that tests the command uses lowercase characters in the string "exit" so DCL determines that the strings are not equal. To correct the error, make sure the quoted string in the IF statement is written in capital letters; the rest of the string can use either upper or lowercase. For example:

```
$     if command .eqs. "EXIT" then goto end_loop
```

5.2.4 Filling in the Program Stubs

When your general design works correctly, complete the command procedure by writing the program stubs. After you add a stub to the command procedure, test it to make sure the procedure works correctly. For a complicated task, you may find it helpful to test the stub before adding it to the general procedure.

The following example shows the code for the TYPE section of CLEAN.COM:

```
$! Execute if user entered TYPE
$ TYPE:
$     IF COMMAND .NES. "TYPE" THEN GOTO ERROR
$     INQUIRE FILE "File to type"
$     TYPE 'FILE'
$     GOTO GET_COMMAND_LOOP
```

5.3 Techniques for Controlling Execution Flow

The previous section illustrated some techniques you can use when designing procedures and directing the flow of execution. This section describes these techniques in more detail.

5.3.1 The IF Command

The IF command tests the value of an expression and executes a given command if the result of the expression is true.

The IF command has the following format:

```
IF expression THEN [$] command
```

An expression is true if:

- It has an odd integer value
- It has a character string value that begins with any of the letters Y, y, T, or t
- It has a character string value that contains numbers that form an integer with an odd value (for example, the string "27")

An expression is false if:

- It has an even integer value.
- It has a character string value that begins with any letter except Y, y, T, or t.
- It has a character string value that contains numbers that form an integer with an even value (for example, the string "28")

The following examples illustrate expressions that can be used with the IF command. For additional examples, see the description of the IF command in the *VAX/VMS DCL Dictionary*. The first example uses a logical operator:

```
$ INQUIRE CONT "Do you want to continue [Y/N]"
$ IF .NOT. CONT THEN EXIT
   .
   .
   .
```

In this example, if the symbol CONT is not true (that is, if the value is not odd or does not begin with Y, y, T, or t) then the procedure exits. Therefore, if the value of CONT is N, the procedure exits; if the value of CONT is Y, then the procedure continues.

The next example uses a symbol as the entire IF expression:

```
$ INQUIRE CHANGE "Do you want to change the record [Y/N]"
$ IF CHANGE THEN GOTO GET_CHANGE
    .
    .
    .
$ GET_CHANGE:
    .
    .
    .
```

In this example, if the symbol CHANGE is true (that is, if the value is odd or begins with Y, y, T, or t) then the procedure goes to the label GET_CHANGE. Otherwise the procedure continues.

The next example illustrates two different IF commands. The first IF command compares two integers; the second IF command compares two strings. Note the difference between the .EQ. and the .EQS. operator.

```
$ COUNT = 0
$ LOOP:
$ COUNT = COUNT + 1
$ IF COUNT .EQ. 9 THEN EXIT
$ IF P'COUNT' .EQS. "" THEN EXIT
    .
    .
    .
```

First, the value of COUNT is compared to the integer 9; if the values are equal, then the procedure exits. If the values are not equal, the procedure continues. This ensures that the loop terminates if eight parameters (the maximum number allowed) have been processed.

In the second IF command, the string value of the symbol P'COUNT' is compared to a null string to see if the symbol is undefined. Note that you must use apostrophes to force iterative substitution. First, the apostrophes cause the value of COUNT to be substituted. Then the IF command performs its usual symbol substitution. For example, If COUNT is 2, the result of the first translation is P2. Then, the value of P2 is used in the string comparison.

Note: When you write an expression for an IF command, follow the rules for writing expressions given in Chapter 2 of this book. (These rules are explained in more detail in the *VAX/VMS DCL Dictionary*.) In particular, remember that:

- **When you use symbols in IF statements, their values are automatically substituted. Do not use apostrophes as substitution operators unless you have a special reason to do so.**
- **String comparison operators end in the letter "S". For example, use operators such as .EQS., .LTS., and .GTS. to compare strings. By contrast, the operators .EQ., .LT., and .GT. are used for comparing integers.**
- **When you test to see whether two strings are equal, the strings must use the same case in order for DCL to find a match. That is, the string "COPY" does not equal the string "copy".**

The target of an IF command can be a single DCL command, a label identifying a block of commands, or another command procedure. In the next example, the target of the IF command is the EXIT command:

```
$ GET_COMMAND_LOOP:
$    INQUIRE COMMAND -
     "Enter command (COPY, DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
$    IF COMMAND .EQS. "EXIT" THEN EXIT
```

The next example uses the GOTO command to direct the flow of execution to a label that identifies a block of commands:

```
$ GET_COMMAND_LOOP:
$    INQUIRE COMMAND -
     "Enter command (COPY, DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
$    IF COMMAND .EQS. "EXIT" THEN GOTO END_LOOP
   .
   .
   .
$ END_LOOP:
   .
   .
   .
```

Instead of placing a group of commands together and starting the group with a label, you can place the commands in a separate command procedure and execute the procedure if the result of the IF expression is true. For example:

```
$ GET_COMMAND_LOOP:
$    INQUIRE COMMAND -
     "Enter command (COPY, DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
$    IF COMMAND .EQS. "EXIT" THEN @EXIT_ROUTINE
```

5.3.2 The GOTO Command

The GOTO command passes control to a labeled line in a command procedure. You can precede any command string in a command procedure with a label. The rules for entering labels are:

- A label must appear as the first item on a line.
- A label can have up to 255 characters.
- No blanks can be embedded within a label.
- A label must be terminated with a colon.

For example:

```
$ GOTO BYPASS
    .
    .
    .
$ BYPASS:
```

As the command interpreter encounters labels, it enters them in a special section of the local symbol table. If a label is encountered that already exists in the table, the new definition replaces the existing one. Note that the amount of space available for labels is limited. If a command procedure uses many symbols and contains many labels, the command interpreter may run out of table space and issue an error message. If this occurs, include the DELETE/SYMBOL command in your procedure to delete symbols as they are no longer needed.

The GOTO command is especially useful within a THEN clause to cause a procedure to branch forward or backward. For example, when you use parameters in a command procedure, you can test the parameters at the beginning of the procedure and branch to the appropriate label:

```
$ IF P1 .NES. "" THEN GOTO OKAY
$ INQUIRE P1 "Enter file spec"
$ OKAY:
$ PRINT/COPIES=10 'P1'
    .
    .
    .
```

In this example, the IF command checks that P1 is not a null string. If P1 is a null string, the GOTO command is not executed and the INQUIRE command prompts for a parameter value. Otherwise, the GOTO command causes a branch around the INQUIRE command. In either case, the procedure executes the PRINT command following the line labeled OKAY.

5.3.3 Loops

Use loops to repeat a group of statements until a condition is met. In general, when you write loops:

1. Begin the loop with a label.
2. Test a variable to determine whether you need to execute the commands in the loop. (This is the termination variable.)
3. If you do not need to execute the loop, go to the end of the loop.
4. If you need to execute the loop, perform the commands in the loop and then return to the beginning of the loop.

You can also write loops that test the termination variable at the end of the loop.

The following examples show different ways you can write loops. The first example tests the termination variable at the beginning of the loop; the loop terminates when the value of the symbol FILE is the null string.

```
$ LOOP:
$    INQUIRE FILE "File (press RET to quit)"
$    IF FILE .EQS. "" THEN GOTO DONE
$    ! Process file
   .
   .
   .
$    GOTO LOOP
$!
$ DONE:
   .
   .
   .
```

In this example, the INQUIRE command requests a file name. If the response is a null value (a CTRL/Z or a RETURN) the loop is not executed. Otherwise, the loop executes, repeatedly, until a null value is entered.

The next example uses a counter to control the number of times a loop is executed. In this example, the loop is executed 10 times; the termination variable is tested at the end of the loop.

```
$! Obtain 10 file names and store them in the
$! symbols FILE_1 through FILE_10
$!
$ COUNT = 0
$ LOOP:
$    COUNT = COUNT + 1
$    INQUIRE FILE_'COUNT' "File"
$    IF COUNT .LT. 10 THEN GOTO LOOP
$!
$ PROCESS_FILES:
   .
   .
   .
```

This example uses the symbol COUNT to keep track of how many times the commands in the loop are executed. The example also uses COUNT to create the symbol names FILE_1, FILE_2 and so on through FILE_10. Note that the value of COUNT is incremented at the beginning of the loop but is tested at the end of the loop. Therefore, when COUNT is incremented from 9 to 10, the loop executes a last time (obtaining a value for FILE_10) before the IF statement finds a false result.

To perform a loop for a known sequence of values, use the F$ELEMENT lexical function. The F$ELEMENT function obtains items from a list of items separated by delimiters. You must supply the item number, the item delimiter, and the list as arguments for F$ELEMENT. See the *VAX/VMS DCL Dictionary* for more information. For example:

```
$ FILE_LIST = "CHAP1/CHAP2/CHAP3/CHAP4/CHAP5"
$ NUM = 0
$!
$! Process each file listed in FILE_LIST
$ PROCESS_LOOP:
$     FILE = F$ELEMENT(NUM,"/",FILE_LIST)
$     IF FILE .EQS. "/" THEN GOTO DONE
$     COPY 'FILE'.MEM MORRIS::DISK3:[DOCSET]*.*
$     GOTO PROCESS_LOOP
$!
$ DONE:
$ WRITE SYS$OUTPUT "Finished copying files."
$ EXIT
```

This command procedure uses a loop to copy the files listed in the symbol FILE_LIST to a directory on another node. The first file returned by the F$ELEMENT function is CHAP1, the next file is CHAP2, and so on. Each time through the loop, the value of NUM is increased so that the next file name is obtained. When the F$ELEMENT returns a slash, then all the items from FILE_LIST have been processed and the loop is terminated.

5.3.4 Case Statements

A case statement is a special form of conditional code that executes one out of a set of command blocks depending on the value of a variable or expression. Although DCL does not provide a case statement, you can perform the function of a case statement in the following way:

1. Equate a symbol to a string that contains a list of options.
2. Obtain the desired option from the user.
3. Use the F$LOCATE and F$LENGTH lexical functions to verify that the user's option is valid; that is, the option is one of the choices listed.
4. Use the GOTO command to direct the flow of execution to the appropriate block of commands. The labels that identify each block of commands should be the same as the options you specify in your option list.

For example, you can rewrite CLEAN.COM (shown in Section 5.2.2) so that it verifies that a command is valid at the beginning of the procedure. If the command is valid, then the procedure executes the command. Note that the labels that identify each command block are the same as the commands in the option list. This allows you to use the symbol COMMAND (which is equated to user's request) in the GOTO statement.

```
$ COMMAND_LIST = "DELETE/DIRECTORY/EXIT/" +
                 "PRINT/PURGE/TYPE/"
$!
$ GET_COMMAND_LOOP:
$    INQUIRE COMMAND -
     "Enter command (DELETE, DIRECTORY, EXIT, PRINT, PURGE, TYPE)"
$    IF F$LOCATE(COMMAND+"/",COMMAND_LIST) .EQ. -
       F$LENGTH(COMMAND_LIST) THEN GOTO ERROR
$    GOTO 'COMMAND'
$!
$!Execute if user entered DELETE
$    DELETE:
$       WRITE SYS$OUTPUT "This is the DELETE section."
$       GOTO GET_COMMAND_LOOP
$!
$!Execute if user entered DIRECTORY
$    DIRECTORY:
$       WRITE SYS$OUTPUT "This is the DIRECTORY section."
$       GOTO GET_COMMAND_LOOP
   .
   .
   .
$!
$ EXIT:
$   WRITE SYS$OUTPUT "Directory ''F$DIRECTORY()' has been cleaned."
$   EXIT
$!
$! Error section
$     ERROR:
$       WRITE SYS$OUTPUT "You entered an invalid command."
$       GOTO GET_COMMAND_LOOP
```

5.4 Terminating Command Procedures

You can use either the EXIT or STOP command to terminate the execution of a command procedure. When included in a command procedure, the EXIT command terminates execution of the current command procedure and returns control to the calling command level. The STOP command, however, returns control to command level 0, regardless of the current command level. If you execute the STOP command in a batch job, the batch job terminates.

You can interrupt a command procedure with a CTRL/Y and then issue either the EXIT or STOP command to terminate the procedure. In this case, both commands return you to command level 0. For example:

```
$ @TESTALL RET
CTRL/Y

$ EXIT RET
$
```

In the above example, the procedure TESTALL is interrupted by CTRL/Y. The EXIT command terminates processing of the procedure and restores command level 0. Note that you could have also issued the STOP command after you interrupted the procedure.

Note: When you interrupt a command procedure, if the command (or image) that you interrupt declared any exit-handling routines, the EXIT command gives these routines control. However, the STOP command does not execute these routines. See Chapter 7 for more information on CTRL/Y interrupts.

5.4.1 Using the EXIT Command

You can use the EXIT command to ensure that a procedure does not execute certain lines. For example, if you write an error-handling routine at the end of a procedure, you would place an EXIT command before the routine, as follows:

```
   .
   .
   .
$      EXIT ! End of normal execution path
$ ERROR_ROUTINE:
   .
   .
   .
```

The EXIT command is also useful for writing procedures that have more than one execution path. For example:

```
$ START:
$       IF P1.EQS. "TAPE" .OR. P1 .EQS. "DISK" THEN GOTO 'P1'
$       INQUIRE P1 "Enter device (TAPE or DISK)"
$       GOTO START
$ TAPE:  !  Process tape files
   .
   .
   .
$      EXIT
$ DISK:  !  Process disk files
   .
   .
   .
$      EXIT
```

To execute this command procedure, you must enter either TAPE or DISK as a parameter. The IF command uses a logical OR to test whether either of these strings was entered. If so, the GOTO command branches appropriately, using the parameter as the branch label. If P1 was neither TAPE nor DISK, the INQUIRE command prompts for a correct parameter; the GOTO START command establishes a loop.

The commands following each of the labels TAPE and DISK provide different paths through the procedure. The EXIT command before the label DISK ensures that the commands after the label DISK are not executed unless the procedure explicitly branches to DISK.

Note that the EXIT command at the end of the procedure is not required because the end-of-file of the procedure causes an implicit EXIT command. Use of the EXIT command, however, is recommended.

5.4.2 Passing Status Values with the EXIT Command

When a command procedure exits, the command interpreter returns the condition code for the previous command in a special symbol called $STATUS. (The condition code provides information about whether the most recent command executed successfully or with errors. See Chapter 7 for more information on condition codes.)

When you use the EXIT command in a command procedure, you can specify a value that overrides the value that DCL would have assigned to $STATUS. This value, called a status code, must be specified as an integer expression.

When a command procedure contains nested procedures to create multiple command levels, you can use the EXIT command to return a value that explicitly overrides the default condition codes.

For example, suppose the procedure A.COM contains:

```
$ @B
   .
   .
   .
```

and the procedure B.COM contains the lines:

```
$ ON WARNING THEN GOTO ERROR
   .
   .
   .
$ ERROR:
$ EXIT 1
```

The ON command means that if any warnings, errors, or severe errors occur when B.COM is executing, the procedure is directed to the label ERROR. Here, the condition code is explicitly set to 1, indicating success. Therefore, when B.COM terminates it will pass a success code back to A.COM regardless of whether an error occurred.

5.4.3 Using the STOP Command

You can use the STOP command in a command procedure or batch job to ensure that all procedures are terminated if a severe error occurs. For example:

```
$ ON SEVERE_ERROR THEN STOP
   .
   .
   .
```

If you include this line in a command procedure and a severe error occurs, then the command procedure terminates. In a command procedure that is executed interactively, control is returned to command level 0. In a batch job, the job terminates.

6 FILE I/O

The basic steps in reading and writing files from a command procedure are:

- Use the OPEN command to open a file. The OPEN command assigns a logical name to the file and specifies whether the file is to be read and/or written. You can open the file for read access, for write access, or for both.
- Use the READ or WRITE commands to read or write records to the file. When you perform input and output to files, you usually design a loop to read a record, process the record, and write the modified record to either the same or to another file. You execute the commands in this loop (reading, processing, and writing records) until you are through.
- Use the CLOSE command to close the file. Unless you explicitly close it, a file remains open until you log out.

This chapter describes how to use the OPEN, READ, WRITE, and CLOSE commands. In addition, it shows how to modify files and how to handle errors when you perform file operations.

6.1 Commands for File I/O

The following sections describe the DCL commands for performing input and output to files.

6.1.1 The OPEN Command

The OPEN command can open sequential, relative, or indexed sequential access method (ISAM) files. When opening a file, the OPEN command assigns a logical name to the file and places the name in the process logical name table. Subsequent READ and WRITE commands use this logical name to refer to the file.

When you open a file from a command procedure, be sure to include the file's device and directory names. This ensures that your command procedure will open the correct file regardless of which directory you execute the procedure from.

The OPEN command opens files as process-permanent. Therefore, these files remain open for the duration of your process unless you explicitly close them (with the CLOSE command). These files are subject to RMS restrictions on using process-permanent files.

You do not have to explicitly open SYS$INPUT, SYS$OUTPUT, SYS$COMMAND, and SYS$ERROR in order to read or write to them because the system opens these files for you when you log in. For more information on reading or writing to these files, see Chapter 3.

The following sections describe how to open a file for reading, writing, and both reading and writing.

6.1.1.1 Opening a File for Reading

To open a file for reading, use the /READ qualifier; this is the default when you use the OPEN command. The following example opens a file for reading:

```
$ OPEN/READ INFILE DISK4:[MURPHY]STATS.DAT
```

When you open a file for reading, you can read, but not write, records. The OPEN/READ command places the record pointer at the beginning of the file. Each time you read a record, the pointer is moved to the next record.

6.1.1.2 Opening a File for Writing

To open a file for writing, use either the /WRITE or the /APPEND qualifier. These qualifiers are mutually exclusive so you can use only one of them on a command line.

The /WRITE qualifier (used without the /READ qualifier) creates a new file and places the record pointer at the beginning of the file. The OPEN/WRITE command always creates a sequential file in print file format. The record format for the file is variable with fixed control (VFC), with a two-byte record header. Note that if you specify a file that already exists, the OPEN/WRITE command opens a new file with a version number one greater than the existing file.

The following example opens a file for writing and writes records to the file:

```
$ OPEN/WRITE OUTFILE DISK4:[MURPHY]NAMES.DAT
$ INQUIRE NEW_RECORD "Enter name"
$ WRITE OUTFILE NEW_RECORD
    .
    .
    .
$ CLOSE OUTFILE
```

The /APPEND qualifier, unlike the /WRITE qualifier, does not open a new version of a file. The /APPEND qualifier opens an existing file and positions the record pointer at the end of the file. This allows you to add records to the end of a file.

The following command procedure appends records to the end of the file NAMES.DAT:

```
$ OPEN/APPEND OUTFILE DISK4:[MURPHY]NAMES.DAT
$ INQUIRE NEW_RECORD "Enter name"
$ WRITE OUTFILE NEW_RECORD
    .
    .
    .
$ CLOSE OUTFILE
```

6.1.1.3 Opening a File for Reading and Writing

To open a file for reading and writing, use both the /READ and /WRITE qualifiers. For example:

```
$ OPEN/READ/WRITE FILE DISK4:[MURPHY]STATS.DAT
```

This OPEN command places the record pointer at the beginning of the file STATS.DAT so you can read the first record. When you use this method of opening a file, you can only replace the record you have most recently read; you cannot write new records to the end of the file. Also, a revised record must be exactly the same size as the record being replaced.

6.1.1.4 Opening Shareable Files

To open a shareable file, use the /SHARE qualifier when you open the file. When you use the /SHARE qualifier, other users may read or write to the file. Use /SHARE=READ to allow other users to read the file; use /SHARE=WRITE to allow other users to write to the file. In addition, other users may access the file with the TYPE or SEARCH command.

6.1.2 The READ Command

Use the READ command to read a record and assign its contents to a symbol. For example:

```
$ OPEN/READ INFILE DISK4:[MURPHY]STATS.DAT
$ READ INFILE RECORD
    .
    .
    .
```

In this example the READ command reads a record from the file INFILE and assigns the record's contents to the symbol RECORD.

When you specify a symbol name for the READ command, the command interpreter places the symbol name in the local symbol table for the current command level. If you use the same symbol name for more than one READ command, each READ command redefines the value of the symbol name. For example:

```
$ BEGIN:
$   READ INFILE RECORD
    .
    .
    .
$   GOTO BEGIN
```

Each time through this loop, the READ command reads a new record from the input file and uses this record to redefine the value of the symbol RECORD.

In order to read from a file, the file must be opened with read access. You can read records that are less than or equal to 1024 characters in length.

6.1.2.1 Designing Read Loops

When you read from files, you generally read and process each record until you reach the end of the file. To determine when you reach the end of a file, use the /END_OF_FILE qualifier with the READ command. By using this qualifier, you can construct a loop to read records from a file, process the records, and exit from the loop when you have finished reading all the records. For example:

```
$ OPEN/READ INFILE DISK4:[MURPHY]STATS.DAT
$ READ_LOOP:
$   READ/END_OF_FILE=END_LOOP INFILE RECORD
        .
        .
        .
$   GOTO READ_LOOP
$!
$ END_LOOP:
$   CLOSE INFILE
$ EXIT
```

In this example, the procedure executes the READ command repeatedly until the end-of-file status is returned. Then the procedure branches to the line labeled END_LOOP. Note that the labels you specify for /END_OF_FILE qualifiers are subject to the same rules as labels specified for a GOTO command.

You should always use the /END_OF_FILE qualifier when you use the READ command in a loop. Otherwise, when the error condition indicating the end-of-file is returned by the VAX Record Management Services (VAX RMS), the command interpreter performs the error action specified by the current ON command. For example, VAX RMS returns the error status

%RMS-E-EOF. This causes a command procedure to exit unless the procedure has established its own error handling.

6.1.2.2 Reading Records Randomly from ISAM Files

You can use the READ command with the /INDEX and /KEY qualifiers to read records randomly from indexed sequential access method (ISAM) files. The /KEY and /INDEX qualifiers specify that a record should be read from the file by finding the specified key in the index, and returning the record associated with that key. If you do not specify an index, the primary index, 0, is used.

Once a record is read randomly, you can read the remainder of the file sequentially from that point by issuing READ commands without the /KEY or /INDEX qualifiers.

You can use the READ command with the /DELETE qualifier to delete records from indexed sequential access method (ISAM) files. The /DELETE qualifier causes a record to be deleted from a file after it has been read. Use the /DELETE qualifier with the /INDEX and /KEY qualifiers to delete a record specified by a given key.

For more information on the /DELETE, /INDEX, and /KEY qualifiers, see the description of the READ command in the *VAX/VMS DCL Dictionary*.

6.1.3 The WRITE Command

Use the WRITE command to write a record to a file. Specify the data to be written as a character string expression, or as a list of expressions. For example:

```
$ OPEN/WRITE OUTFILE DISK4:[MURPHY]NEW_STATS.DAT
$ WRITE OUTFILE "File created April 15, 1984"
    .
    .
    .
$ CLOSE OUTFILE
```

When you specify data for the WRITE command, follow the rules for character string expressions described in Chapter 2. Note that the WRITE command automatically substitutes symbols and lexical functions, as shown below:

```
$ OPEN/WRITE OUTFILE DISK4:[MURPHY]NEW_STATS.DAT
$ INQUIRE RECORD "Enter name"
$ WRITE OUTFILE NAME
   .
   .
   .
```

In this example, the WRITE command writes the value of the symbol NAME to NEW_STATS.DAT.

You can intersperse symbol names and literal character strings within a WRITE command, as shown below:

```
$ WRITE OUTFILE "Count is ",COUNT,"."
```

If the value of COUNT is 4, this WRITE command writes the following string to the file OUTFILE:

```
Count is 4.
```

Another way to mix literal strings with symbol names is to place the entire string within quotation marks and use double apostrophes to request symbol substitution. For example:

```
$ WRITE OUTFILE "Count is ''COUNT'."
```

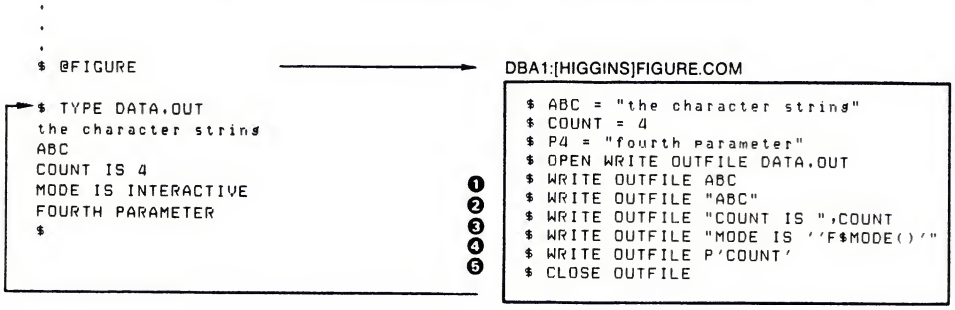
Figure 6–1 shows different ways of specifying data for the WRITE command.

The WRITE command can write records only to files that have been opened for writing. (You can use either the /WRITE or the /APPEND qualifiers to open files for writing.) When the WRITE command writes a record, it positions the record pointer after the record just written. The WRITE command can write a record that is up to 2048 bytes long.

If you wish to write a record that is longer than 1024 bytes, or if any expression in the WRITE command is longer than 255 bytes, then you must use the /SYMBOL qualifier when you write the record. See the description of the WRITE command in the *VAX/VMS DCL Dictionary* for more information on writing long records.

You can use the WRITE command with the /UPDATE qualifier to change a record rather than insert a new one. In order to use the /UPDATE qualifier, you must have opened a file for both reading and writing. (See Section 6.2.1.)

Figure 6–1 Symbol Substitution with the WRITE Command



❶ The WRITE command automatically performs symbol substitution on characterstrings that are not enclosed in quotation marks; substitution is not recursive.

❷ If a character string is enclosed in quotation marks, the WRITE command does not perform symbol substitution.

❸ When two or more symbol names or character strings are specified, the WRITE command concatenates the strings before it writes the record to the output file.

❹ Within character strings, the command interpreter performs substitution requested by apostrophes during command input; the WRITE command executes the results.

❺ If the data specified for a WRITE command contains an apostrophe, the command interpreter performs symbol substitution during command input (as in ❹); the WRITE command performs substitution on the resulting command string.

ZK-830-82

6.1.4 The CLOSE Command

The CLOSE command closes a file and deassigns the logical name created by the OPEN command. For example:

```
$ OPEN INFILE DISK4:[MURPHY]STATS.DAT
    .
    .
    .
$ CLOSE INFILE
```

Whenever you open a file in a command procedure, be sure that the file is closed before the command procedure terminates. If you fail to close an open file, then the file remains open, and the logical name assigned to the file is not deleted from the logical name table.

6.2 Modifying a File

There are three ways you can modify files:

- Update records in the current version of the file.
- Read records from an input file and create a new version of the file that incorporates your changes.
- Append records to the current version of the file.

The following sections describe these methods.

6.2.1 Updating Records in a File

When you update a file, you can replace existing records although you cannot add new records. When you update a file, you do not create a new version of the input file.

In a sequential file, you can update individual records only if the revised records are exactly the same size as the existing records. In an ISAM file, you can update individual records regardless of the record sizes.

To update records in a file:

1. Open the file for both read and write access.
2. Use the READ command to obtain the record(s) that you want to modify.
3. Modify the record. In a sequential file, the text of this record must be exactly the same size as the original record. If the text of the modified record is shorter, pad the record with spaces. If the text of the modified record is longer, you cannot use this method to modify the file.
4. Use the WRITE command with the /UPDATE qualifier to write the modified record back to the file.
5. Continue using the READ and WRITE commands to read and update the records in the file until you have finished updating the file.
6. Use the CLOSE command to close the file.

After you close the file, it contains the same version number as when you started even though individual records have been changed.

The following command procedure shows how to make changes to a sequential file by reading and updating individual records.

```
$! Open STATS.DAT and assign it the logical name FILE
$!
$ OPEN/READ/WRITE FILE DISK4:[MURPHY]STATS.DAT
$ BEGIN_LOOP:
$! Read the next record from FILE into the symbol RECORD
$     READ/END_OF_FILE=END_LOOP FILE RECORD
$! Display the record and see if the user wants to change it
$! If yes, get the new record.  If no, repeat loop
$     PROMPT:
$          WRITE SYS$OUTPUT RECORD
$          INQUIRE/NOPUNCTUATION YN "Change? Y or N [Y] "
$          IF YN .EQS. "N" THEN GOTO BEGIN_LOOP
$          INQUIRE NEW_RECORD "New record"
$! Compare the old and new records
$! If old record is shorter than new record, issue an error message
$! If old record and new record are the same length, write the record
$! Otherwise pad new record with spaces so it is correct length
$          OLD_LEN = F$LENGTH(RECORD)
$          NEW_LEN = F$LENGTH(NEW_RECORD)
$          IF OLD_LEN .LT. NEW_LEN THEN GOTO ERROR
$          IF OLD_LEN .EQ. NEW_LEN THEN GOTO WRITE_RECORD
$          SPACES = "                "
$          PAD = F$EXTRACT(0,OLD_LEN-NEW_LEN,SPACES)
$          NEW_RECORD = NEW_RECORD + PAD
$!
$     WRITE_RECORD:
$          WRITE/UPDATE FILE NEW_RECORD
$          GOTO BEGIN_LOOP
$!
$     ERROR:
$          WRITE SYS$OUTPUT "Error---New record is too long"
$          GOTO PROMPT
$!
$     END_LOOP:
$          CLOSE FILE
$          EXIT
```

6.2.2 Creating a New Output File

To make extensive changes to a file, you may need to rewrite the entire file. Because you are creating a new output file, you can modify the size of records, add records, or delete records.

To create a new output file, follow these steps:

1. Use the OPEN command to open a file for read access. This is the input file, the file you are modifying.
2. Use the OPEN command to open a new file for write access. This is the output file, the file that you are creating. If you give the output file the same name as the input file, the output file will have a version number one greater than the input file.
3. Use the READ command to read a record from the file you are modifying.
4. If the record needs to be changed, modify the record and then use the WRITE command to write the record to the new file. If the record is correct, then write it directly to the new file. If the record is to be deleted, do not write it to the new file.
5. Continue reading and processing records until you have finished.
6. Use the CLOSE command to close both the input and the output files.

The following example shows a command procedure that reads a record from an input file, processes the record, and copies the record into an output file.

```
$! Open STATS.DAT for reading and assign it
$! the logical name INFILE
$! Open a new version of STATS.DAT for writing
$! and assign it the logical name OUTFILE
$!
$ OPEN/READ INFILE DISK4:[MURPHY]STATS.DAT
$ OPEN/WRITE OUTFILE DISK4:[MURPHY]STATS.DAT
$!
$ BEGIN_LOOP:
$! Read the next record from INFILE into the symbol RECORD
$     READ/END_OF_FILE=END_LOOP INFILE RECORD
$! Display the record and see if the user wants to change it
$! If yes, get the new record.  If no, write record directly
$! to OUTFILE
$     PROMPT:
$          WRITE SYS$OUTPUT RECORD
$          INQUIRE/NOPUNCTUATION YN "Change? Y or N [Y] "
$          IF YN .EQS. "N" THEN GOTO WRITE_RECORD
$          INQUIRE RECORD "New record"
$!
$     WRITE_RECORD:
$          WRITE OUTFILE RECORD
$          GOTO BEGIN_LOOP
$!
$! Close input and output files
$     END_LOOP:
$          CLOSE INFILE
$          CLOSE OUTFILE
$          EXIT
```

6.2.3 Appending Records to a File

The OPEN/APPEND command allows you to append records to the end of an existing file. To append records, follow these steps:

1. Use the OPEN command with the /APPEND qualifier to position the record pointer at the end of the file. The /APPEND qualifier does not create a new version of the file.
2. Use the WRITE command to write new data records. Continue adding records until you are through.
3. Use the CLOSE command to close the file.

The following example appends new records to a file.

```
$! Open STATS.DAT to append files and assign
$! it the logical name FILE
$!
$ OPEN/APPEND FILE DISK4:[MURPHY]STATS.DAT
$!
$ BEGIN_LOOP:
$! Obtain record to be appended and place this
$! record in the symbol RECORD
$     PROMPT:
$          INQUIRE RECORD -
           "Enter new record (press RET to quit) "
$          IF RECORD .EQS. "" THEN GOTO END_LOOP
$! Write record to FILE
$          WRITE FILE RECORD
$          GOTO BEGIN_LOOP
$!
$! Close FILE and exit
$     END_LOOP:
$          CLOSE FILE
$          EXIT
```

6.3 Handling I/O Errors

Use the /ERROR qualifier with the OPEN, READ, WRITE, and CLOSE commands to handle errors. The /ERROR qualifier directs control to a label in your command procedure if an I/O error occurs. Also, the /ERROR qualifier suppresses error messages that would normally be displayed. The following example uses the /ERROR qualifier with the OPEN command:

```
$ OPEN/READ/ERROR=CHECK FILE CONTINGEN.DOC
    .
    .
    .
$ CHECK:
$  WRITE SYS$OUTPUT "Error opening file"
```

The OPEN command requests that the file CONTINGEN.DOC be opened for reading. If the file cannot be opened (for example, if the file does not exist) the OPEN command returns an error condition. Control is then transferred to the label CHECK.

The error path specified by the /ERROR qualifier overrides the current ON condition established for the command level. If an error occurs and the target label is successfully given control, the reserved global symbol $STATUS retains the code for the error. You can use the F$MESSAGE lexical function in your error-handling routine to display the message in $STATUS:

```
$ OPEN/READ/ERROR=CHECK FILE 'P1'
    .
    .
    .
$ CHECK:
$  ERR_MESSAGE = F$MESSAGE($STATUS)
$  WRITE SYS$OUTPUT "Error opening file: ",P1
$  WRITE SYS$OUTPUT ERR_MESSAGE
    .
    .
    .
```

If an error occurs during an OPEN, READ, WRITE, or CLOSE command and you did not specify an error action, then the current ON command action is taken. (See Chapter 7.)

When a READ command receives an end of file message, the error action is determined in the following way. If you used the /END_OF_FILE qualifier, then control is passed to the specified label. If /END_OF_FILE is not specified, then control is given to the label specified with the /ERROR qualifier. If neither qualifier is specified, the current ON action is taken.

7 Controlling Error Conditions and CTRL/Y Interrupts

This chapter describes what happens when an error condition or a CTRL/Y interrupt occurs while a command procedure is executing. It also describes how you can use the ON, SET [NO]ON, and SET [NO]CONTROL commands to change the defaults for handling error conditions and CTRL/Y interrupts.

7.1 Detecting Errors in Command Procedures

When each DCL command in a command procedure completes execution, the command interpreter saves a condition code that describes the reason why the command terminated. This code can indicate successful completion or it can identify a specific informational or error message.

The command interpreter examines the condition code after it performs each command in a command procedure. If an error that requires special action has occurred, the system performs the action. Otherwise, the next command in the procedure is executed.

7.1.1 Condition Codes and $STATUS

The condition code is saved as a 32 bit longword in the reserved global symbol $STATUS. $STATUS conforms to the format of a VAX/VMS message code:

- Bits 0–2 contain the severity level of the message.
- Bits 3–15 contain the message number.
- Bits 16–27 contain the number associated with the facility that generated the message.
- Bits 28–31 contain internal control flags.

When a command completes successfully, $STATUS has an odd value. (Bits 0–2 contain a 1 or a 3.) When any type of warning or error occurs, $STATUS has an even value. (Bits 0–2 contain a 0, 2, or 4.) The command interpreter maintains and displays the current value of $STATUS in hexadecimal.

7.1.2 Severity Levels and $SEVERITY

The low-order three bits of $STATUS represent the severity of the condition that caused the command to terminate. This portion of the condition code is contained in the reserved global symbol $SEVERITY. $SEVERITY can have the values 0 through 4, with each value representing one of the following severity levels:

Value	Severity
0	Warning
1	Success
2	Error
3	Information
4	Fatal error

Note that the success and information codes have odd numeric values and warning and error codes have even numeric values. You can test for the successful completion of a command with IF commands that perform logical tests on $SEVERITY or $STATUS as shown below:

```
$ IF $SEVERITY THEN GOTO OKAY
$ IF $STATUS THEN GOTO OKAY
```

These IF commands branch to the label OKAY if $SEVERITY and $STATUS have true (odd) values. When the current value in $SEVERITY and $STATUS is odd, the command or program completed successfully. If the command or program did not complete successfully, then $SEVERITY and $STATUS are even; therefore the IF expression is false.

Instead of testing that a condition is true, you can test whether it is false. For example:

```
$ IF .NOT. $STATUS THEN ...
```

The command interpreter uses the severity level of a condition code to determine whether to take the action defined by the ON command as described in Section 7.2.1.

7.1.3 Commands That Do Not Set $STATUS

Most DCL commands invoke system utilities that generate status values and error messages when they complete. However, there are several commands that do not change the values of $STATUS and $SEVERITY if they complete successfully. These commands are:

CONTINUE	IF
DECK	SHOW STATUS
DEPOSIT	SHOW SYMBOL
EOD	STOP
EXAMINE	WAIT
GOTO	

If any of these commands results in a nonsuccessful status, however, that condition code will be placed in $STATUS, and the severity level will be placed in $SEVERITY.

7.2 Error Condition Handling

By default, the command interpreter executes an EXIT command when a command results in an error or severe error. This causes the procedure to exit to the previous command level. For other severity levels (success, warning, and informational) the command procedure continues.

There is one exception to the way that the command interpreter handles errors. If you reference a label in a command procedure and the label does not exist (for example, if you include the command GOTO ERR1 and ERR1 is not used as a label in the procedure) then the GOTO command issues a warning and the command procedure exits. You can, however, override this behavior if you use the ON command to specify an action for

the command procedure to take on warnings. (Section 7.2.1 describes the ON command.)

When the system issues an EXIT command as part of an error handling routine, it passes the value of $STATUS back to the previous command level, with one change. The command interpreter sets the high-order digit of $STATUS to 1 so that the command interpreter does not redisplay the message associated with the status value.

For example, the following command procedure TEST.COM contains an error in the output file specification:

```
$ CREATE DUMMY.DAT\
THIS IS A TEST FILE
$ SHOW TIME
```

When you execute the procedure, the CREATE command returns an error in $STATUS and displays the corresponding message. The command interpreter then examines the value of $STATUS, determines that an error occurred, issues an EXIT command, and returns the value of $STATUS. When the procedure exits, the error message is not redisplayed, as the CREATE command already displayed the message once. Back at DCL command level you can see that $STATUS contains the error message but the high-order digit has been set to 1.

```
$ @TEST
%CREATE-E-OPENOUT, error opening DUMMY.DAT
$ SHOW SYMBOL $STATUS
  $STATUS = "%X109110A2"
$ WRITE SYS$OUTPUT F$MESSAGE(%X109110A2)
  %CREATE-E-OPENOUT, error opening !AS as output
```

You can use the ON and SET [NO]ON commands to change the way that the system responds to errors in command procedures.

7.2.1 The ON Command

The ON command specifies an action to be performed if an error of a particular severity or worse occurs. If such an error occurs, the system takes the following actions:

- The action specified by the ON command is performed.
- The system sets $STATUS and $SEVERITY to indicate the result of the specified ON action. (In general, these are set to success.)

- The default error action (to exit if an error or severe error occurs) is reset.

The format of the ON command is:

```
ON condition THEN [$] command
```

You can specify error conditions with one of the keywords WARNING, ERROR, or SEVERE_ERROR. If an ON command action is established for a specific severity level, the command interpreter will perform the specified action when errors of the same or worse severity occur. When less severe errors occur, the command interpreter will continue processing the file. Table 7–1 summarizes how the ON command controls error handling.

Table 7–1 ON Command Keywords and Actions

ON Keyword	Action Taken
WARNING	Command procedure performs the specified action if a warning, error, or severe error occurs.
ERROR	Command procedure performs the specified action if an error or severe error occurs; the procedure continues if a warning occurs.
SEVERE_ERROR	Command procedure performs the specified action if a severe (fatal) error occurs; the procedure continues if a warning or error occurs.

For example, if you want to override the default error handling so that a procedure will exit when warnings, errors, or severe errors occur, use the command:

```
$ ON WARNING THEN EXIT
```

An ON command action is executed only once; thus, after a command procedure performs the action specified in an ON command, the default error action is reset. For example, suppose that your procedure includes the following command:

```
$ ON ERROR THEN GOTO ERR1
```

In this case, the command procedure executes normally until an error or severe error occurs. If such an error occurs, then the procedure resumes executing at ERR1, $STATUS and $SEVERITY are set to success, and the default error action is reset. If a second error occurs before another ON or SET NOON command is executed, the procedure exits to the previous command level.

The action specified by an ON command applies only within the command procedure in which the command is executed. Therefore, if you execute an ON command in a procedure that calls another procedure, the ON command action does not apply to the nested procedure. In fact, an ON command executed at any command procedure level does not affect the error condition handling of procedures at any other level.

Figure 7–1 illustrates ON command actions. Also, the sample procedures FORTUSER.COM and CALC.COM in Appendix A illustrate the use of the ON command to establish error handling.

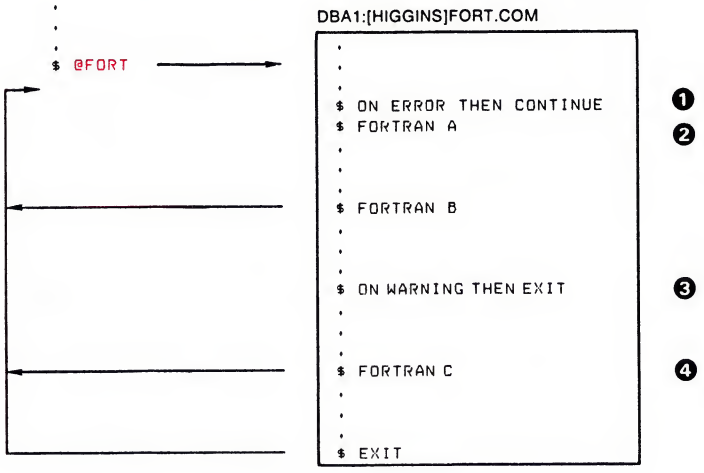
7.2.2 Disabling Error Checking

You can prevent the command interpreter from checking the status returned from commands by issuing the SET NOON command in your command procedure. When you issue the SET NOON command, the command interpreter continues to place values in $STATUS and $SEVERITY, but does not perform any error checking. You can restore error checking with the SET ON command or with an ON command. For example:

```
$ SET NOON
$ RUN TESTA
$ RUN TESTB
$ SET ON
```

The SET NOON command preceding these RUN commands ensures that the command procedure will continue if either of the programs TESTA or TESTB return an error condition. The SET ON command restores the default error checking by the command interpreter.

Figure 7–1 ON Command Actions



1. This ON command overrides the default command action (on warning, continue; on error or severe error, exit). If an error or severe error occurs while A.FOR is being compiled, the command procedure continues with the next command.
2. The default command action is reset if the previous ON command takes effect. Thus, if an error or severe error occurs while both A.FOR and B.FOR are being compiled, the command procedure exits.
3. If the command procedure does not exit before this command is executed, this command action takes effect.
4. If a warning, error, or severe error occurs while C.FOR is being compiled, the command procedure exits.

ZK-826-82

When a procedure disables error checking, it can explicitly check the value of $STATUS following the execution of a command or program. For example:

```
$ SET NOON
$ FORTRAN MYFILE
$ IF $STATUS THEN LINK MYFILE
$ IF $STATUS THEN RUN MYFILE
$ SET ON
```

In the above example, the first IF command checks whether $STATUS has a true value (that is, if it is an odd numeric value). If so, the FORTRAN command was successful and the LINK command will be executed. After the LINK command, $STATUS is tested again. If $STATUS is odd, the RUN command will be executed; otherwise, the RUN command will not be executed. The SET ON command restores the current ON condition action; that is, whatever condition was in effect before the SET NOON command was executed.

The SET ON or SET NOON command applies only at the current command level, that is, the command level at which the command is executed. If you use the SET NOON command in a command procedure that calls another command procedure, the default error checking will be in effect within the nested procedure. Note that SET NOON has no meaning when issued interactively at command level 0.

7.3 Handling CTRL/Y Interrupts

By default, when you press CTRL/Y while a command procedure is executing, the command interpreter prompts for command input at a special command level called CTRL/Y command level. From CTRL/Y command level, you can issue DCL commands that are executed within the command interpreter, and then resume execution of the command procedure with the CONTINUE command. In addition, you can stop the procedure by executing a DCL command that forces the command procedure to stop executing.

You can override the way that command procedures process CTRL/Y interrupts by using the ON command as described in Section 7.3.2.

7.3.1 Interrupting a Command Procedure

You can interrupt a command procedure that is executing interactively by pressing CTRL/Y. When you press CTRL/Y, the command interpreter establishes a new command level, called the CTRL/Y level, and prompts for command input. When the interruption actually occurs depends on the command that is executing:

- If the command currently executing is a command that is executed by the command interpreter itself (for example, IF, GOTO, or an assignment statement) the command completes execution before the command interpreter prompts for a command at the CTRL/Y level.
- If the command or program currently executing is a separate image (that is, an image other than the command interpreter), the command is interrupted and the command interpreter prompts for a command at the CTRL/Y level.

At the CTRL/Y level, the command interpreter stores the status of all previously established command levels, so that it can restore the correct status after any CTRL/Y interrupt.

After you interrupt a procedure, you can:

- Issue a DCL command that is executed within the command interpreter. Among these commands are the SET VERIFY, SHOW TIME, SHOW TRANSLATION, ASSIGN, EXAMINE, DEPOSIT, SPAWN and ATTACH commands. After you issue one or more of these commands, you can resume the execution of the procedure with the CONTINUE command. (See the *VAX/VMS DCL Dictionary* for a complete list of commands that are executed within the command interpreter.)

  When you issue the CONTINUE command the command procedure resumes execution with the interrupted command or program, or with the line after the most recently completed command.

- Issue a DCL command that executes another image. When you issue any command that invokes a new image, the command interpreter returns to command level 0 and executes the command. This terminates the command procedure's execution. Any exit handlers declared by the interrupted image will be allowed to execute before the new image is started.

- Issue the EXIT or STOP command to terminate the command procedure's execution. If you use the EXIT command, exit handlers declared by the interrupted image will be allowed to execute. However, the STOP command does not execute these routines.

If you interrupt the execution of a privileged image, you can issue only the CONTINUE, SPAWN, or ATTACH commands if you want to save the context of the image. If you issue any other commands (except from within a subprocess that you have spawned or attached to) the privileged image is forced to exit.

7.3.2 Setting a CTRL/Y Action Routine

The ON command, which defines an action to be taken in case of error conditions, also provides a way to define an action routine for a CTRL/Y interrupt that occurs during execution of a command procedure. The action that you specify overrides the default action of the command interpreter (that is, to prompt for command input at the CTRL/Y command level).

For example:

```
$ ON CONTROL_Y THEN EXIT
```

If a procedure executes the ON command shown above, a subsequent CTRL/Y interrupt during the execution of the procedure causes the procedure to exit. Control is passed to the previous command level.

When you press CTRL/Y to interrupt a procedure that uses ON CONTROL_Y, the following actions are taken:

- If the command currently executing is a command executed within the command interpreter, the command completes and then the CTRL/Y action is taken.
- If the current command is executed by an image other than the command interpreter, the image is forced to exit and then the CTRL/Y action is taken. If the image has declared an exit handler, however, the exit handler is executed before the CTRL/Y action is taken. The image cannot be continued following the CTRL/Y action.

The execution of a CTRL/Y action does not automatically reset the command procedure default CTRL/Y action. A CTRL/Y action remains in effect until:

- The procedure terminates (as a result of an EXIT or STOP command, or a default error condition handling action)
- Another ON CONTROL_Y command is executed
- The procedure executes the SET NOCONTROL=Y command (See Section 7.3.3.)

For example, a procedure can contain the line:

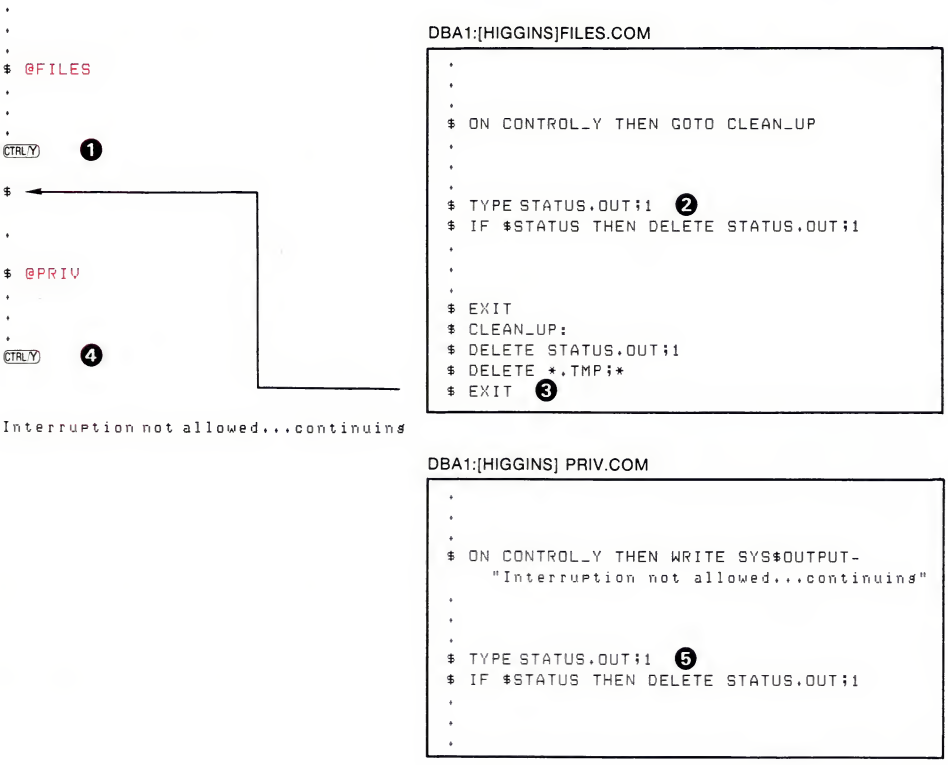
```
$ ON CONTROL_Y THEN SHOW TIME
```

When this procedure executes, each CTRL/Y interrupt results in the execution of the SHOW TIME command. After each SHOW TIME command executes, the procedure resumes execution at the command following the command that was interrupted.

Figure 7–2 illustrates two ON CONTROL_Y commands and describes the flow of execution following CTRL/Y interruptions.

A CTRL/Y action can be specified for each active command level; the CTRL/Y action specified for the currently executing command level overrides action(s) specified for previous levels, if any. Note, however, that if a CTRL/Y action is established at a command level, the default action for subsequent command levels is to exit. Figure 7–3 illustrates what happens when CTRL/Y is pressed during the execution of a nested command procedure.

Figure 7–2 Flow of Execution Following CTRL/Y Action

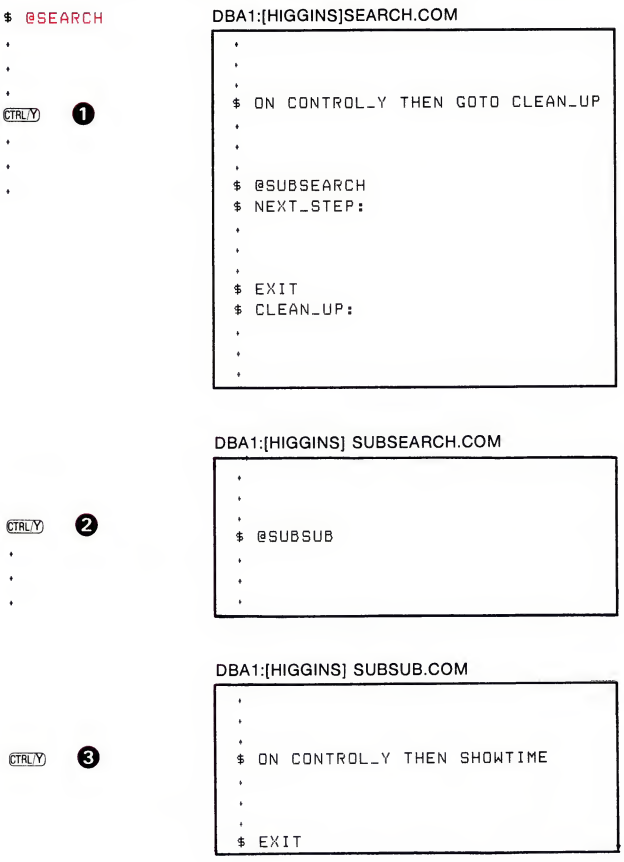


The CTRL/Y interrupt at 1 occurs during execution of the TYPE command, at 2. Control is transferred to the label CLEAN_UP. After executing the routine, the command procedure exits, at 3 and returns control to the interactive command level.

The CTRL/Y interrupt at 4 occurs during execution of the TYPE command, at 5. The WRITE command specified in the ON command is executed. Then, the command procedure continues execution at the command following the interrupted command.

ZK-827-82

Figure 7–3 Default CTRL/Y Action for Nested Procedures



1. If a CTRL/Y interrupt occurs while SEARCH.COM is executing, control is transferred to the label CLEAN__UP.
2. If a CTRL/Y interrupt occurs while SUBSEARCH.COM is executing, control is transferred to the label NEXT__STEP in SEARCH.COM. Because no CTRL/Y action is specified in SUBSEARCH.COM, the procedure exits to previous command level when a CTRL/Y interrupt occurs.
3. If a CTRL/Y interrupt occurs while SUBSUB.COM is executing, the SHOW TIME command is executed.

ZK-828-82

7.3.3 Disabling and Enabling CTRL/Y Interruptions

The SET NOCONTROL=Y command disables CTRL/Y handling completely: that is, if a command procedure executes the SET NOCONTROL=Y command, pressing CTRL/Y will have no effect.

The SET NOCONTROL=Y command also cancels the current CTRL/Y action established with the ON CONTROL_Y command. To reestablish the default command interpreter action for CTRL/Y handling, issue the following two commands:

```
$ SET NOCONTROL=Y
$ SET CONTROL=Y
```

The first command disables CTRL/Y handling and cancels the current ON CONTROL_Y action; the second command enables CTRL/Y handling. At this point, the default action is reinstated: if CTRL/Y is pressed during the execution of the procedure, the command interpreter prompts for a command at the CTRL/Y command level.

You can issue the SET NOCONTROL=Y command at any command level; it affects all command levels until the SET CONTROL=Y command reenables CTRL/Y handling.

Note: The ON CONTROL_Y and SET NOCONTROL=Y commands are intended for special applications. DIGITAL does not recommend, in general, that you disable CTRL/Y interrupts. For example, if a procedure in which CTRL/Y interrupts are disabled begins to loop uncontrollably, the procedure can be terminated by another process that stops the process from which the looping procedure is executing.

8 Working with Batch Jobs

A batch job consists of one or more command procedures that you execute from another process. To execute a batch job, use the SUBMIT command to place the job in a batch queue. (A batch queue is a list of batch jobs waiting to execute.) After the system places your job in the batch queue, you can continue using your terminal interactively. Thus, a batch job allows you to use your terminal for other work while the system executes your command procedure.

When the system executes your job, it creates a detached process for you. To create a detached process, the system logs you in using the information from your UAF record. The system executes your login command file, and then executes the command procedure(s) in the batch job. As these procedures execute, output is written to a log file. When the job is complete, you can print the log file and/or save it in one of your directories.

In general, you use batch jobs to execute command procedures that either take a long time to execute, that you want to schedule for execution after a specific time, or that use a disproportionate level of system resources. (For those procedures that are likely to use a high level of system resources, you can specify that the batch job be executed at a reduced priority.)

This chapter describes techniques for submitting and controlling batch jobs. The chapter discusses:

- Submitting batch jobs
- Controlling batch job output
- Controlling jobs in a batch queue
- Restarting batch jobs
- Synchronizing batch jobs

8.1 Submitting a Batch Job

Use the SUBMIT command to enter a command procedure into a batch queue. By default, the SUBMIT command places command procedures into a queue named SYS$BATCH. If you omit the file type when you submit a command procedure, the type defaults to COM. For example:

```
$ SUBMIT UPDATE
Job UPDATE (queue SYS$BATCH, entry 201) started on SYS$BATCH
```

This command creates a job called UPDATE that contains the command procedure UPDATE.COM. This job is entered in the queue SYS$BATCH and is assigned the entry number 201. (After a job has been submitted, you usually refer to it using the entry number.) The system message shows that SYS$BATCH has started to execute your job. Your batch job executes as if you had logged in and executed the commands in the command procedure UPDATE.COM.

When you submit a command procedure for batch execution, the system saves the complete file specification for the command procedure, including the version number. If you update a command procedure after you submit it, the batch job still executes the original version of your command procedure.

Since your login defaults are not usually the defaults needed to access files mentioned in your command procedures, use one of the following methods to ensure that the correct files are accessed:

- Use complete file specifications—When referring to a file in a command procedure or when passing a file to a command procedure, include the device and directory names as part of the file specification.
- Use the SET DEFAULT command—Before referring to a file in a command procedure, use the SET DEFAULT command to specify the proper device and directory.

As a batch job executes, output is written to a log file. By default, the log file has the same name as the command procedure you submit, with a file type of LOG. When the job is finished, the system prints the log file and then deletes it from your directory. See Section 8.3.1 for information on saving log files.

8.1.1 Checking for Batch Jobs in Your Login Command File

Each time you submit a batch job, the system executes your login command file. You can cause sections of your login command file to be included or omitted when you execute batch jobs by using the F$MODE() lexical function to test for batch jobs.

For example, you might have a section in your login command file that includes commands, logical names, and symbols that you use exclusively for batch jobs. You might label this section BATCH_COMMANDS, then include the following command at the beginning of your login command file:

```
IF F$MODE() .EQS. "BATCH" THEN GOTO BATCH_COMMANDS
   .
   .
   .
```

To prevent the system from executing any commands in your login command file, place the following command either at the beginning of the file:

```
IF F$MODE() .NES. "INTERACTIVE" THEN EXIT
```

You can place this command anywhere in your login command file. When you submit a batch job, the system executes your login command file only to the point at which the EXIT command is placed.

8.1.2 Submitting Multiple Command Procedures

When you issue the SUBMIT command, you can specify several command procedures to be executed in one job. For example:

```
$ SUBMIT UPDATE,SORT
Job UPDATE (queue SYS$BATCH, entry 207) started on SYS$BATCH
```

This SUBMIT command creates a batch job that executes UPDATE.COM, then SORT.COM. Unless you specify a name with the /NAME qualifier, the SUBMIT command uses the name of the first command procedure as the job name. Note that if an error causes any command procedure in a job to exit, the entire job terminates.

When a batch job executes, the operating context of the first procedure (UPDATE.COM) is not preserved for the second procedure (SORT.COM). That is, the system deletes local symbols created by UPDATE.COM before SORT.COM executes. Global symbols, however, are preserved.

You cannot specify different parameters for individual command procedures within a single job. The following example passes the same two parameters to UPDATE.COM and SORT.COM:

```
$ SUBMIT UPDATE, SORT/PARAMETERS = -
_$ (DISK1:[ACCOUNT.BILLS]DATA.DAT, DISK2:[ACCOUNT]NAME.DAT)
  Job UPDATE (queue SYS$BATCH, ENTRY 208) started on SYS$BATCH
```

8.1.3 Controlling the Batch Job

You can control how a batch job is submitted by using the appropriate qualifiers with the SUBMIT command. The following list describes some common operations and the qualifiers you use to perform them. For a complete list of qualifiers, see the *VAX/VMS DCL Dictionary*.

You can do the following when you submit a batch job:

- Name the job—Use the /NAME qualifier to specify a name for the batch job. Otherwise the job name defaults to the file name of the first (or only) command procedure in the job.
- Specify the time when the batch job starts to execute—Use the /AFTER qualifier to specify a time after which the job can be executed. When you use /AFTER, the job remains in the batch queue until the specified time; then it is executed. To hold a job in the queue until you explicitly release it, use the /HOLD qualifier. (To release a job that is being held, use the SET QUEUE/ENTRY/RELEASE command.)
- Request notification of job completion—Use the /NOTIFY qualifier to have the system send a message to your terminal when the batch job finishes executing.
- Send the job to a specific queue—Use the /QUEUE qualifier to send a batch job to a queue other than SYS$BATCH. To execute a command procedure that is located on a remote node, use the /REMOTE qualifier. This sends the job to SYS$BATCH at the remote node.

- Specify execution characteristics—You can specify execution characteristics such as working set default, working set extent, working set size, job scheduling priority, and CPU time limit. See the *VAX/VMS DCL Dictionary* for more information.
- Pass parameters—Use the /PARAMETERS qualifier to pass parameters to a batch job.
- Save the log file—Use the /NOPRINT or /KEEP qualifiers to save a batch job log file. See Section 8.3 for information on controlling batch job output.
- Make the batch job restartable—Use the /RESTART qualifier to enable you to restart the job if the system fails while the job is executing. See Section 8.5 for information on restarting batch jobs.

If you submit jobs frequently using the same qualifiers, place a symbol for the SUBMIT command string in your login file. For example:

```
$ SUBMIT == "SUBMIT/NOTIFY/NOPRINT"
```

8.2 Passing Data to Batch Jobs

When you execute batch jobs, the default input stream (SYS$INPUT) is the command procedure that is being executed. Because the batch job is being executed from a detached process, you cannot redefine SYS$INPUT to the terminal (as you can with interactive command procedures.) Therefore, pass input to a batch job in one of the following ways:

- Include the data in the command procedure itself
- Temporarily define SYS$INPUT as a file
- Pass parameters to the command procedure

To include data in a command procedure, place the data on the lines after the command or image. For example:

```
$! Execute AVERAGE.EXE
$ RUN AVERAGE
647
899
532
401
$ EXIT
```

To temporarily define SYS$INPUT as a file, use the DEFINE /USER_MODE command. For example:

```
$ DEFINE/USER_MODE SYS$INPUT STATS.DAT
$ RUN AVERAGE
$ EXIT
```

To pass parameters to a command procedure, use the /PARAMETERS qualifier when you submit the batch job. For example:

```
$ SUBMIT/PARAMETERS=(DISK1:[PAYROLL]EMPLOYEES.DAT) CHECKS
Job CHECKS (queue SYS$BATCH, entry 209) started on SYS$BATCH
```

Note that you cannot specify different parameters for individual command procedures within a single job; use separate SUBMIT commands if you need to pass different groups of parameters.

Note: The SHOW QUEUE/FULL command displays full information about jobs in a batch queue. This display includes any parameters you pass to the procedure. Therefore, do not pass confidential information (such as a password) to a batch job.

8.3 Batch Job Output

If you use the SUBMIT command without any qualifiers that change the log file, the system creates a log file in your default login directory. This log file has the same name as the first command procedure in the batch job, and a file type of LOG. Once each minute, output from the batch job is written to the log file. When the job finishes, the log file is printed on the default system printer and it is deleted from your directory.

The batch job log file includes all output to SYS$OUTPUT and SYS$ERROR. It also includes, by default, all command lines executed in the command procedure. To prevent the command lines from being printed, use either the SET NOVERIFY command or the F$VERIFY lexical function in your command procedure. When the job completes, the system writes job termination information (using the long form of the system logout message) to the log file.

When a batch job fails to complete successfully, you can examine the log file to determine the point at which the command procedure failed and the error status.

8.3.1 Saving the Log File

To save the log file, use either the /KEEP or the /NOPRINT qualifier. The /KEEP qualifier saves the log file after it is printed; the /NOPRINT qualifier saves the log without printing it. These qualifiers save the log file in your default login directory; to specify an alternate file and/or directory name, use the /LOG_FILE qualifier. To rename and save the log file, you must use /LOG_FILE plus either /KEEP or /NOPRINT. For example:

```
$ SUBMIT/LOG_FILE=DISK2:[JONES.RESULTS]/NOPRINT -
_$ DISK2:[JONES.RESULTS]UPDATE
```

8.3.2 Reading the Log File

You can use the TYPE command to read the log file to determine how much of a batch job has completed. However, if you attempt to display the log file while the system is writing to it, you receive a message indicating that the file is locked by another user. If this occurs, wait a few seconds and try again.

Note that the system writes output to the log file once each minute. To specify a different time interval, include the SET OUTPUT_RATE command in your command procedure.

8.3.3 Including All Command Output in the Batch Job Log

Typically, a batch job that compiles, links, and executes a program creates additional printed output such as a compiler listing or a linker map. To produce printed copies of these files, a batch job can contain the PRINT command(s) necessary to print them, as in the following example:

```
$ FORTRAN BIGCOMP
$ PRINT BIGCOMP
$ LINK/MAP/FULL BIGCOMP
$ PRINT BIGCOMP.MAP
```

When this batch job completes processing, there are three separate output listings: the batch job log, the compiler listing, and the linker map.

If you want a batch job log to contain all output from the command procedure, including printed listings of compiler or linker output files, you can do either of the following:

- Use the TYPE command instead of the PRINT command in the command procedure. The TYPE command writes to SYS$OUTPUT. (In a batch job, SYS$OUTPUT is equated to the batch job log file.)
- Use qualifiers on appropriate commands to direct the output to SYS$OUTPUT.

The following example shows the latter technique:

```
$ FORTRAN/LIST=SYS$OUTPUT  BIGCOMP
$ LINK/MAP=SYS$OUTPUT/FULL BIGCOMP
```

When these commands are executed in a batch job, the output files from the compiler and the linker are written directly to the log file. Note that if you use this technique, the output file(s) are not saved on disk.

8.4 Controlling Jobs in a Batch Queue

Once a job has been entered in a batch job queue, you can monitor its status with the SHOW QUEUE command. For example:

```
$ SUBMIT/NOPRINT/PARAMETER=STATS.DAT UPDATE
$ SHOW QUEUE SYS$BATCH
Batch queue SYS$BATCH, on BOSTON::
  Jobname         Username     Entry  Status
  -------         --------     -----  ------
  UPDATE          ODONNELL      1080  Executing
```

First, the output shows the queue and node names. Then the job name, user name, entry number, and status are displayed. If you had no jobs in the queue, you would have received the following message:

```
$ SHOW QUEUE SYS$BATCH
Batch queue SYS$BATCH, on BOSTON::
```

To see complete information on your jobs, use the /FULL qualifier:

```
$ SHOW QUEUE/FULL SYS$BATCH
Job UPDATE (queue SYS$BATCH, entry 1080) started on SYS$BATCH
Batch queue SYS$BATCH, on BOSTON::
    /BASE_PRIORITY=3 /JOB_LIMIT=5 /OWNER=[EXEC] /PROTECTION=(S:E,O:D,G:R,W:W)
  Jobname         Username     Entry  Status
  -------         --------     -----  ------
  UPDATE          ODONNELL      1080  Executing
    Submitted 15-FEB-1984 10:46 /KEEP /PARAM=("STATS.DAT") /NOPRINT /PRIO=4
      _BOSTON$DQA2:[ODONNELL]TEMP.COM;1 (executing)
```

The /FULL qualifier displays statistics about SYS$BATCH. This qualifier also displays the characteristics associated with your job.

To see the status of other jobs in the queue, use the SHOW QUEUE/ALL command. For example:

```
SHOW QUEUE/ALL SYS$BATCH
Batch queue SYS$BATCH, on BOSTON::
  Jobname         Username     Entry  Status
  -------         --------     -----  ------
  no privilege                   923  Executing
  no privilege                   939  Executing
  UPDATE          ODONNELL      1080  Executing
```

Note that unless you are a privileged user, you may be able to obtain information only about jobs that have been submitted under your account. See the *VAX/VMS DCL Dictionary* for more information on the SHOW QUEUE command.

8.4.1 Changing Job Characteristics

After a job has been submitted to the queue but before the job starts to execute, you can use the SET QUEUE/ENTRY command with the appropriate qualifiers to change characteristics associated with the job. For example, to change the name of a batch job while it is pending in a batch queue, you can issue the following command:

```
$ SET QUEUE/ENTRY=209/NAME=NEW_NAME SYS$BATCH
```

This command changes the name of the job number 209 to NEW_NAME.

Some of the changes you can make with SET QUEUE/ENTRY are described below; for a complete list of qualifiers see the *VAX/VMS DCL Dictionary*. Note that most of the qualifiers allowed with the SUBMIT command can also be used with SET QUEUE/ENTRY.

You can make the following changes:

- Delay processing of a job—Use the /AFTER qualifier to specify a time after which the job can be executed; use the /HOLD qualifier to hold a job until you explicitly release it.
- Release a job—Use the /NOHOLD or /RELEASE qualifier to release a job that was submitted with the /HOLD or /AFTER qualifiers.
- Send a job to a different queue—Use the /REQUEUE qualifier to change the queue on which the job will execute.
- Change execution characteristics—You can change execution characteristics such as working set default, working set extent, working set size, job scheduling priority, and CPU time limit. See the *VAX/VMS DCL Dictionary* for more information.
- Change the parameters to be passed to a job—Use the /PARAMETERS qualifier to change the parameters.

8.4.2 Deleting and Stopping Batch Jobs

You can delete batch jobs before or during execution. To delete an entry that is pending or already executing in a batch queue, use the DELETE/ENTRY command. For example, the following command deletes a job in SYS$BATCH:

```
$ DELETE/ENTRY=210 SYS$BATCH
```

You need special privileges to delete a job that you did not submit. See the *VAX/VMS DCL Dictionary* for more information.

When a job terminates as a result of a DELETE/ENTRY command, the log file is neither printed nor deleted from your directory.

When you terminate a job using the DELETE/ENTRY command, it is handled as an abnormal termination because the operating system's normal job termination activity is preempted. As a result, the batch job log does not, for example, contain the standard logout message that summarizes job time and accounting information. However, termination that results either from an explicit EXIT or STOP command in the procedure or the implicit execution of either of these commands following an error condition based on the current ON condition

is considered normal termination, and the operating system performs proper run-down and accounting procedures.

8.5 Restarting Batch Jobs

If the system fails while your batch job is executing, your job does not complete. When the system recovers and the queue is restarted, your job is aborted and the next job in the queue is executed. However, by specifying the /RESTART qualifier when you submit a job, you indicate that the system should reexecute the job if the system crashes before the job completes.

By default, a batch job is restarted beginning with the first line. However, you can specify a different starting point so that you do not reexecute parts of the job that have successfully completed. To do this, follow these steps:

- Begin each possible starting point in the procedure with a label. After the label, use the SET RESTART_VALUE command to set the restarting point to that label. If the batch job is interrupted by a system crash and is then restarted, the SET RESTART_VALUE command assigns the appropriate label name to the global symbol BATCH$RESTART.
- At the beginning of the procedure, test the value of the symbol $RESTART. ($RESTART is a global symbol that the system maintains for you. $RESTART has the value "TRUE" if one of your batch jobs was restarted after it was interrupted by a system crash. Otherwise, $RESTART has the value "FALSE".) If $RESTART is true, issue a GOTO statement using BATCH$RESTART as the transfer label.

The following command procedure shows how to use restart values in a batch job.

```
$ ! set default to the directory containing
$ ! the file to be updated and sorted
$ SET DEFAULT DISK1:[ACCOUNTS.DATA84]
$
$ ! check for restarting
$ IF $RESTART THEN GOTO 'BATCH$RESTART'
$
$ UPDATE_FILE:
$ SET RESTART_VALUE = UPDATE_FILE
    .
    .
    .
$ SORT_FILE:
$ SET RESTART_VALUE = SORT_FILE
    .
    .
    .
EXIT
```

To submit this command procedure as a batch job that can be restarted, use the /RESTART qualifier when you submit the job. If the job is interrupted by a system crash and then restarted, the job will start executing in the section where it was interrupted. For example, if the job is interrupted during the SORT_FILE routine, it will start executing at the label SORT_FILE when it is restarted.

In addition to restarting a job after a system crash, you can also restart a job after you explicitly stop the job. To stop a job and then restart it on the same or a different queue, use the STOP/QUEUE/REQUEUE/ENTRY command. For example:

```
$ STOP/QUEUE/REQUEUE/ENTRY=212 SYS$BATCH
```

This command stops job 212 on SYS$BATCH, and re-queues it on SYS$BATCH. (To issue this command, job 212 must have been submitted with the /RESTART qualifier.) When job 212 executes the second time, the system will use BATCH$RESTART to determine where to begin executing the job.

8.6 Synchronizing Batch Job Execution

You can use the SYNCHRONIZE and WAIT commands within a command procedure to place the procedure in a wait state. The SYNCHRONIZE command causes the procedure to wait for the completion of a specified job, while the WAIT command causes the procedure to wait for a specified period of time to elapse. For example, if two jobs are submitted concurrently to perform cooperative functions, one job can contain the command:

```
$ SYNCHRONIZE BATCH25
```

After this command is executed, the command procedure cannot continue execution until the job identified by the job name BATCH25 completes execution. Figure 8–1 shows an example of command procedures that are submitted for concurrent execution, but which must be synchronized for proper execution. Each procedure compiles a large source program.

Figure 8–1 Synchronizing Batch Job Execution

```
.
.
.
$ SUBMIT MAINCOMP
  JOB 314 entered on queue SYS$BATCH (1)
$ SUBMIT MINCOMP
  JOB 315 entered on queue SYS$BATCH (1)
.
.
.
```

DBA1:[HIGGINS]MAINCOMP.COM

```
$ FORTRAN ALPHA/LIST
$ SYNCHRONIZE MINCOMP
$ LINK/MAP/FULL ALPHA,BETA
$ EXIT
```

(2)

DBA1:[HIGGINS] MINCOMP.COM

```
$ FORTRAN BETA/LIST
$ EXIT
```

(1) Individual SUBMIT commands are required to submit two separate jobs. Two separate processes will be created.

(2) After the FORTRAN command is executed, the SYNCHRONIZE command is executed. If job 315 has completed execution, job 314 continues with the next command. However, job 314 will not execute the next command, if job 315 is either current or pending.

ZK-832-82

Job names specified for the SYNCHRONIZE command must be for jobs that are executing with the same group number in their user identification codes (UICs). To synchronize with a job that

has a different group number (for example, that was submitted by a different user), you must use the job entry number. For example:

```
$ SYNCHRONIZE/ENTRY=454
```

This SYNCHRONIZE command places the current command procedure in a wait state until job 454 completes.

The WAIT command is useful for command procedures that must have access to a shared system resource, for example, a disk or tape drive. The following example shows a procedure that requests the allocation of a tape drive; if the command does not complete successfully, the procedure will place itself in a wait state. After a five-minute interval, it retries the request:

```
$ TRY:
$       ALLOCATE DM: RK:
$       IF $STATUS THEN GOTO OKAY
$       WAIT 00:05
$       GOTO TRY
$ OKAY:
$ REQUEST/REPLY/TO=DISKS -
     "Please mount BACK_UP_GMB on ''F$TRNLNM("RK")'"
          .
          .
          .
```

The IF command following the ALLOCATE request checks the value of $STATUS. If the value of $STATUS indicates successful completion, the command procedure will continue. Otherwise, the procedure issues the WAIT command; the WAIT command specifies a time interval of five minutes. After waiting five minutes, the next command, GOTO, is executed, and the request is repeated. This procedure continues looping and attempting to allocate a device until it succeeds or until the batch job is deleted or stopped.

A Annotated Command Procedures

This appendix contains complete command procedures that demonstrate the concepts and techniques discussed in this book. Each section in this appendix discusses one command procedure and contains the following:

- The name of the procedure
- A listing of the procedure
- Notes that explain concepts or techniques used by the procedure
- The results of a sample execution of the procedure

The command procedures are:

CONVERT.COM

The sample command procedure CONVERT.COM is shown in Section A.1. This procedure converts an absolute time (for a time in the future) to a delta time. Therefore, the procedure determines the time between the current time and the time that you specify. The procedure illustrates use of the F$TIME and F$CVTIME lexical functions and the use of assignment statements to perform arithmetic calculations and to concatenate symbol values.

REMINDER.COM

The sample command procedure REMINDER.COM is shown in Section A.2. This procedure displays a reminder message on your terminal at a specified time. The procedure prompts for the time you want the message to be displayed, and for the text of the message. The procedure uses CONVERT.COM to convert the time to a delta time. Then the procedure spawns a subprocess that waits till the specified time and then displays your reminder message. The procedure illustrates the use of the F$ENVIRONMENT, F$VERIFY, and F$GETDVI functions.

DIR.COM

The sample command procedure DIR.COM is shown in Section A.3. This procedure imitates the DCL command DIRECTORY /SIZE=ALL/DATE, displaying the block size (used and allocated) and creation date of the specified files. It illustrates use of the F$PARSE, F$SEARCH, F$FILE_ATTRIBUTES, and F$FAO lexical functions.

SYS.COM

The sample command procedure SYS.COM is shown in Section A.4. This procedure returns statistics about processes in the current process list. If the current process has GROUP privilege, statistics for all processes in the group are returned. Statistics for all processes on the system are displayed if the current process has WORLD privilege. This procedure illustrates use of the FPID, FEXTRACT, and F$GETJPI lexical functions.

GETPARMS.COM

The sample command procedure GETPARMS.COM is shown in Section A.5. This procedure returns the number of parameters that were passed to a procedure. GETPARMS.COM can be called from another procedure to determine how many parameters were passed to the calling procedure.

EDITALL.COM

The sample command procedure EDITALL.COM is shown in Section A.6. This procedure invokes the EDT editor repeatedly to edit a group of files with the same file type. This procedure illustrates how to use lexical functions to extract file names from columnar output. It also illustrates a way to redefine the input stream for a program invoked within a command procedure.

FORTUSER.COM

The sample command procedure FORTUSER.COM is shown in Section A.7. This example of a system-defined login file provides a controlled terminal environment for an interactive user who creates, compiles, and executes FORTRAN programs. If a user logs into a captive account where FORTUSER.COM is listed as the login command file, then the user can execute only the commands accepted by FORTUSER.COM. This procedure also illustrates using lexical functions to step through an option table, comparing a user-entered command with a list of valid commands.

LISTER.COM

The sample command procedure LISTER.COM is shown in Section A.8. This is a procedure that prompts for input data, formats the data in columns, and then sorts it into an output file. This procedure illustrates the READ and WRITE commands, as well as the character substring overlay format of an assignment statement.

CALC.COM

The sample command procedure CALC.COM is shown in Section A.9. This procedure performs arithmetic calculations and converts the resulting value to hexadecimal and decimal values.

BATCH.COM

The sample command procedure BATCH.COM is shown in Section A.10. This procedure accepts a command string, a command procedure, or a list of commands and then executes these commands as a batch job.

A.1 CONVERT.COM

```
$ ! Procedure to convert an absolute time to a delta time.
$ ! The delta time is returned as the global symbol WAIT_TIME.
$ ! P1 is the time to be converted.
$ ! P2 is an optional parameter - SHOW - that causes the
$ ! procedure to display WAIT_TIME before exiting


$ !
$ ! Check for inquiry
$ !
$ IF P1 .EQS. "?" .OR. P1 .EQS. "" THEN GOTO TELL     ❶


$ !
$ ! Verify the parameter:  hours must be less than 24
$ !                        minutes must be less than 60
$ !                        time string must contain only hours
$ !                        and minutes
$ !
$ ! Change error and message handling to
$ ! use message at BADTIME
```

```
$ !
$ ON WARNING THEN GOTO BADTIME
$ SAVE_MESSAGE = F$ENVIRONMENT("MESSAGE")
$ SET MESSAGE/NOFACILITY/NOIDENTIFICATION/NOSEVERITY/NOTEXT
$ TEMP = F$CVTIME(P1)


$ !
$ ! Restore default error handling and message format
$ ON ERROR THEN EXIT
$ SET MESSAGE'SAVE_MESSAGE'


$ !
$ IF F$LENGTH(P1) .NE. 5 .OR -
       F$LOCATE(":",P1) .NE. 2 -
       THEN GOTO BADTIME


$ !
$ ! Get the current time
$ !
$ TIME = F$TIME()
$ !
$ ! Extract the hour and minute fields from both the current time
$ ! value (TIME) and the future time (P1)


$ !
$ MINUTES = F$CVTIME(TIME,"ABSOLUTE","MINUTE")       ! Current minutes
$ HOURS = F$CVTIME(TIME,"ABSOLUTE","HOUR")           ! Current hours
$ FUTURE_MINUTES = F$CVTIME(P1,"ABSOLUTE","MINUTE")  ! Minutes in future time
$ FUTURE_HOURS = F$CVTIME(P1,"ABSOLUTE","HOUR")      ! Hours in future time


$ !
$ !
$ ! Convert both time values to minutes
$ ! Note the implicit string to integer conversion being performed


$ !
$ CURRENT_TIME = HOURS*60 + MINUTES
$ FUTURE_TIME = FUTURE_HOURS*60 + FUTURE_MINUTES


$ !
$ ! Compute difference between the future time and the current time
$ ! (in minutes)
$ !
$ !
$ MINUTES_TO_WAIT = FUTURE_TIME - CURRENT_TIME
```

```
$ !
$ ! If the result is less than 0 the specified time is assumed to be
$ ! for the next day; more calculation is required.
$ !
$ IF MINUTES_TO_WAIT .LT. 0 THEN -                    ❽
        MINUTES_TO_WAIT = 24*60 + FUTURE_TIME - CURRENT_TIME


$ !
$ ! Start looping to determine the value in hours and minutes from
$ ! the value expressed all in minutes


$ !
$       HOURS_TO_WAIT = 0
$ HOURS_TO_WAIT_LOOP:                    ❾
$       IF MINUTES_TO_WAIT .LT. 60 THEN GOTO FINISH_COMPUTE
$       MINUTES_TO_WAIT = MINUTES_TO_WAIT - 60
$       HOURS_TO_WAIT = HOURS_TO_WAIT + 1
$       GOTO HOURS_TO_WAIT_LOOP
$ FINISH_COMPUTE:


$ !
$ ! Construct the delta time string in the proper format
$ !
$ WAIT_TIME == F$STRING(HOURS_TO_WAIT)+ ":" + F$STRING(MINUTES_TO_WAIT)-    ❿
  + ":00.00"


$ !
$ ! Examine the second parameter
$ !
$ IF P2 .EQS.  "SHOW" THEN SHOW SYMBOL WAIT_TIME          ⓫


$ !
$ ! Normal exit


$ !
$ EXIT
$ !
$ BADTIME:      ⓬
$ ! Exit taken if first parameter is not formatted correctly
$ ! EXIT command returns but does not display error status


$ !
$ SET MESSAGE'SAVE_MESSAGE'
$ WRITE SYS$OUTPUT  "Invalid time value: ",P1,", format must be hh:mm"
$ WRITE SYS$OUTPUT "Hours must be less than 24; minutes must be less than 60"
$ EXIT %X10000000
```

```
$ !
$ !
$ TELL:       ⓭
$ ! Display message and exit if user enters inquiry or enters
$ ! an illegal parameter


$ !
$ TYPE SYS$INPUT
        This procedure converts an absolute time value to
        a delta time value.  The absolute time must be in
        the form hh::mm and must indicate a time in the future.

        On return, the global symbol WAIT_TIME contains the
        converted time value.  If you enter the keyword SHOW
        as the second parameter, the procedure displays the
        resulting value in the output stream. To invoke this
        procedure, use the following syntax:

                        @CONVERT hh:mm [SHOW]
$ EXIT
```

Notes

❶ The procedure checks whether the parameter was omitted or whether the value entered for a parameter is the question mark character (?). In either case, the procedure will branch to the label TELL.

❷ The procedure uses the F$CVTIME function to verify that the time value is a valid 24-hour clock time; the F$CVTIME returns a warning message if the input time is not valid. Therefore, the procedure changes the default ON action to direct control to the label BADTIME if the F$CVTIME function returns an error.

The procedure uses the F$ENVIRONMENT function to save the current message setting, and then sets the message format so that no warning or error messages are displayed. After checking the time values, the procedure restores the default ON condition and message format.

❸ The procedure checks the format of the parameter. It must be a time value in the format:

```
hh:mm
```

The IF command checks (1) that the length of the entered value is 5 characters and (2) that the third character (offset of 2) is a colon. The IF command contains the logical OR operator: if either expression is true (that is, if the length is not 5 or if there is not a colon in the third character position), the procedure will branch to the label BADTIME.

❹ The F$TIME lexical function places the current time value in the symbol TIME.

❺ The F$CVTIME function extracts the "minute" and "hour" fields from the current time (saved in the symbol TIME). Then the F$CVTIME function extracts the "minute" and "hour" fields from the time you want to convert.

❻ These assignment statements convert the current and future times to minutes. When you use the symbols MINUTES, HOURS, FUTURE_HOURS, and FUTURE_MINUTES in the assignment statements, the system automatically converts these values to integers.

❼ The procedure then subtracts the current time (in minutes) from the future time (in minutes).

❽ If the result is less than 0, the future time is interpreted as being on the next day. In this case, the procedure adds 24 hours to the future time, and then subtracts the current time.

❾ The procedure enters a loop in which it calculates, from the value of MINUTES_TO_WAIT, the number of hours. Each time through the loop, it checks whether MINUTES_TO_WAIT is greater than 60. If so, it will subtract 60 from MINUTES_TO_WAIT and add 1 to the accumulator for the number of hours (HOURS_TO_WAIT).

❿ When the procedure exits from the loop, it concatenates the hours and minutes values into a time string. The symbols HOURS_TO_WAIT and MINUTES_TO_WAIT are replaced by their character string equivalents and separated with an intervening colon. The resulting string is assigned to the symbol WAIT_TIME, which holds the delta time value for the future time. WAIT_TIME is defined as a global symbol so that it will not be deleted when the procedure WAIT_TIME exits.

⓫ If a second parameter, SHOW, was entered, the procedure will display the resulting time value. Otherwise, it will exit.

⓬ At the label BADTIME, the procedure displays an error message that shows the incorrect value entered as well as the format it requires. After issuing the error message, CONVERT.COM exits. The EXIT command returns an error status in which the high order digit is set to 1. This suppresses the display of an error message.

The procedure explicitly specifies an error status with the EXIT command so you can execute CONVERT.COM from within another procedure. When CONVERT.COM completes, the calling procedure can determine whether a time was successfully translated.

⓭ At the label TELL, the procedure displays information about what the procedure does. The TYPE command displays the lines listed in SYS$INPUT, the input data stream.

Sample Execution

```
$ SHOW TIME
10-JUN-1984 10:38:26
$ @CONVERT 12:00 SHOW
   WAIT_TIME = "1:22:00.00"
```

The SHOW TIME command displays the current date and time. CONVERT.COM is executed with the parameters 12:00 and SHOW. The procedure converts the absolute time 12:00 to a delta time value and displays it on the terminal.

A.2 REMINDER.COM

```
$ ! Procedure to obtain a reminder message and display this
$ ! message on your terminal at the time you specify.


$ !
$ ! Save current states for procedure and image verification
$ ! Turn verification off for duration of procedure
$


$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")     ❶
$ SAVE_VERIFY_PROC = F$VERIFY(0)
```

```
$ !
$ ! Places the current process in a wait state until a specified
$ ! absolute time.  Then, it rings the bell on the terminal and
$ ! displays a message.


$ !
$ ! Prompt for absolute time
$ !
$


$ GET_TIME:
$ INQUIRE REMINDER_TIME  "Enter time to send reminder (hh:mm)"        ❷
$ INQUIRE MESSAGE_TEXT "Enter message"


$ !
$ ! Call the CONVERT.COM procedure to convert the absolute time
$ ! to a delta time


$ !
$ @DISK2:[JONES.TOOLS]CONVERT 'REMINDER_TIME'        ❸
$ IF .NOT. $STATUS THEN GOTO BADTIME
$ !


$ !
$ ! Create a command file that will be executed
$ ! in a subprocess.  The subprocess will wait until
$ ! the specified time and then display your message
$ ! at the terminal.  If you are working at a DEC_CRT
$ ! terminal, the message has double size blinking
$ ! characters. Otherwise, the message has normal letters.
$ ! In either case, the terminal bell rings when the
$ ! message is displayed.
$


$ CREATE WAKEUP.COM              ❹
$ DECK                               ! Lines starting with $ are data lines
$ WAIT 'WAIT_TIME'               ❺
$ IF F$GETDVI("SYS$OUTPUT","TT_DECCRT") .NES. "TRUE" THEN GOTO OTHER_TERM
$ BELL[0,7] = %X07                   ! Create symbol to ring the bell


$ !
$ DEC_CRT_ONLY:
$ ! Create symbols to set special graphics (for DEC_CRT terminals only)
```

```
$ !
$     SET_FLASH = "<ESC>[1;5m"     ! Turn on blinking characters
$     SET_NOFLASH = "<ESC>[0m"     ! Turn off blinking characters
$     TOP = "<ESC>#3"              ! Double size characters (top portion)
$     BOT = "<ESC>#4"              ! Double size characters (bottom portion)


$ !
$ ! Write double size, blinking message to the terminal and ring the bell
$ !
$     WRITE SYS$OUTPUT BELL, SET_FLASH, TOP, MESSAGE_TEXT
$     WRITE SYS$OUTPUT BELL, BOT, MESSAGE_TEXT
$     WRITE SYS$OUTPUT F$TIME(), SET_NOFLASH
$     GOTO CLEAN_UP


$ !
$ OTHER_TERM:
$      WRITE SYS$OUTPUT BELL,MESSAGE_TEXT
$      WRITE SYS$OUTPUT F$TIME()


$ !
$ CLEAN_UP:
$      DELETE WAKEUP.COM;*
$ EOD


$ !
$ ! WAKEUP.COM has been created.  Now continue executing commands.
$ !
$ SPAWN/NOWAIT/INPUT=WAKEUP.COM     ❻
$ END:       ❼
$ !   Restore verification
$     SAVE_VERIFY_PROC = F$VERIFY(SAVE_VERIFY_PROC, SAVE_VERIFY_IMAGE)
$     EXIT
$ !
$ BADTIME:
$    WRITE SYS$OUTPUT "Time must be entered as hh:mm"
$    GOTO GET_TIME
```

Notes

❶ The procedure uses the F$ENVIRONMENT function to save the image verification setting in the symbol SAVE_VERIFY_IMAGE. Next, the procedure uses the F$VERIFY function to save the procedure verification setting in the symbol SAVE_VERIFY_IMAGE. The F$VERIFY function also turns both types of verification off.

❷ The procedure uses the INQUIRE command to prompt for the time when the reminder message should be sent. This value will be used as input to the procedure CONVERT.COM. The procedure also prompts for the text of the message.

❸ The procedure executes a nested procedure, CONVERT.COM. Be sure to specify the disk and directory as part of the file specification; this ensures that the system can locate CONVERT.COM regardless of the directory from which you execute REMINDER.COM.

CONVERT.COM converts your reminder to a delta time, and returns this time in the global symbol WAIT_TIME. This delta time indicates the time interval from the current time until the time when the message should be sent. If CONVERT.COM returns an error, the procedure branches to the label BADTIME.

❹ The procedure uses the CREATE command to create a new procedure, WAKEUP.COM. This procedure will be executed from within a subprocess. To allow the CREATE command to read lines that begin with dollar signs, use the DECK and EOD commands to surround the input for the CREATE command. Therefore, all lines between the DECK and EOD commands will be written to WAKEUP.COM.

❺ WAKEUP.COM performs the following tasks:

- It waits until the time indicated by the symbol WAIT_TIME.
- It creates the symbol BELL to ring the terminal bell.
- It determines whether the terminal is a DEC_CRT terminal and can accept escape sequences to display double size, blinking characters. (To see whether you have a DEC_CRT terminal, issue the SHOW TERMINAL command and see whether this characteristic is listed.)
- If the terminal is a DEC_CRT terminal, then the procedure defines the symbols SET_FLASH, TOP, and BOT. These symbols cause the terminal to use flashing, double-size characters. The procedure also defines the symbol SET_NOFLASH to return the terminal to its normal state. To enter the escape character (<ESC>) when you create these definitions using the EDT editor, press the ESC key twice.

After defining these symbols, the procedure writes three lines to the terminal. The first line rings the bell, turns on flashing characters, and displays (using double size characters) the top half of your message. The second line rings the bell again, and displays the bottom half of your message. The third line writes the current time and then turns off the flash characteristic to return your terminal to normal.

If you do not have a DEC_CRT terminal, then the procedure rings your terminal bell, and displays your message and the time.

- The DELETE command causes the procedure WAKEUP.COM to delete itself after it executes.

❻ After creating WAKEUP.COM, the procedure spawns a subprocess and directs the subprocess to use WAKEUP.COM as the input command file. The /NOWAIT qualifier allows you to continue working at your terminal while the subprocess executes commands from WAKEUP.COM. At the specified time, WAKEUP.COM displays your message on your terminal.

Note that, by default, the SPAWN command passes global and local symbols to a subprocess. Therefore, although you provide values for the symbols WAIT_TIME and MESSAGE_TEXT in REMINDER.COM, WAKEUP.COM can also access these symbols.

❼ The procedure restores the original verification settings before it exits.

Sample Execution

```
$ @REMINDER
Enter time to send reminder (hh:mm): 12:00
Enter message: TIME FOR LUNCH
%DCL-S-SPAWNED, process BLUTO_1 spawned
$
   .
   .
   .
TIME FOR LUNCH
15-APR-1984 12:00:56.99
```

The procedure prompts for a time value and for your message. Then the procedure spawns a subprocess that will display your message. You can continue working at your terminal; at the specified time, the subprocess will ring the terminal bell, display your message, and display the time.

A.3 DIR.COM

```
$ !
$ ! Command procedure implementation of DIRECTORY/SIZE=ALL/DATE
$ ! command


$ !
$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(0)
$ P1 = F$PARSE(P1,"*.*;*")              ! Create directory wild      ❶
$ !                                     ! card spec
$ FIRST_TIME = "TRUE"                   ! Header not printed yet
$ FILE_COUNT = 0                        ! No files found yet
$ TOTAL_ALLOC = 0                       ! No blocks allocated yet
$ TOTAL_USED = 0                        ! No blocks used yet
$


$ LOOP:                 ❷
$       FILESPEC = F$SEARCH(P1)
$ ! Find next file in directory
$       IF FILESPEC .EQS. "" THEN GOTO DONE
$ ! If no more files, then done
$       IF .NOT. FIRST_TIME THEN GOTO SHOW_FILE
$ ! Print header only once
$ !
$ ! Construct and output header line
$ !
$       FIRST_TIME =  "FALSE"                ❸
$       DIRSPEC = F$PARSE(FILESPEC,,, "DEVICE") -
                +F$PARSE(FILESPEC,,, "DIRECTORY")
$       WRITE SYS$OUTPUT ""
$       WRITE SYS$OUTPUT  "Directory ",DIRSPEC
$       WRITE SYS$OUTPUT ""
$       LASTDIR = DIRSPEC
$


$ !
$ ! Put the file name together, get some of the file attributes, and
$ ! type the information out
```

```
$ !
$SHOW_FILE:
$       FILE_COUNT = FILE_COUNT + 1
$       FILENAME = F$PARSE(FILESPEC,,, "NAME") -          ❹
                 + F$PARSE(FILESPEC,,, "TYPE") -
                 + F$PARSE(FILESPEC,,, "VERSION")
$       ALLOC = F$FILE_ATTRIBUTES(FILESPEC, "ALQ")
$       USED = F$FILE_ATTRIBUTES(FILESPEC, "EOF")
$       TOTAL_ALLOC = TOTAL_ALLOC + ALLOC
$       TOTAL_USED = TOTAL_USED + USED
$       REVISED = F$FILE_ATTRIBUTES(FILESPEC,"RDT")
$       LINE = F$FAO("!19AS !5UL/!5<!UL!> !17AS",FILENAME,-
                USED, ALLOC, REVISED)
$       WRITE SYS$OUTPUT LINE
$       GOTO LOOP
$


$ !
$ ! Output summary information, reset verification, and exit
$ !
$ DONE:                 ❺
$       WRITE SYS$OUTPUT ""
$       WRITE SYS$OUTPUT "Total of ''FILE_COUNT' files, " + -
                         "''TOTAL_USED'/''TOTAL_ALLOC' blocks."
$       SAVE_VERIFY_PROCEDURE = F$VERIFY(SAVE_VERIFY_PROCEDURE,SAVE_VERIFY_IMAGE)
$       EXIT
```

Notes

❶ This procedure uses the F$PARSE function to place asterisks in blank fields in P1, the user-supplied file specification. If you do not specify a parameter when you execute DIR.COM, then the F$PARSE function assigns the value "*.*;*" to P1. This causes DIR.COM to display all files in the current default directory.

❷ The F$SEARCH lexical function searches the directory for the file (or files) indicated by P1. If P1 contains any wildcards (asterisks), the F$SEARCH function returns all matching file specifications. After the last file specification has been returned, the procedure branches to the label DONE.

❸ The first time through the loop, the procedure writes a header for the directory display. This header includes the device and directory names. To obtain these names, the procedure uses the F$PARSE function.

❹ The procedure uses the F$PARSE lexical function to extract the file name from each file specification in the directory. The F$FILE_ATTRIBUTES lexical function then obtains blocks used, blocks allocated, and creation date information about each file. Finally, the F$FAO function formats a single display line for each file in the directory. The F$FAO function uses the file name and file attribute information as arguments.

❺ When F$SEARCH returns a null string, the procedure branches to the label DONE and summary information is displayed showing total number of files the total blocks used, and the total blocks allocated in the directory.

Sample Execution

```
$ @DIR [VERN]*.COM
  Directory DISK4:[VERN]

  BATCH.COM;1              1/3      16-JUN-1984 11:43
  CALC.COM;3               1/3      16-JUN-1984 11:30
  CONVERT.COM;1            5/6      16-JUN-1984 15:23
    .
    .
    .
  LOGIN.COM;34             2/3      16-JUN-1984 13:17
  PID.COM;7                1/3      16-JUN-1984 09:49
  SCRATCH.COM;6            1/3      16-JUN-1984 11:29

  Total of 15 files, 22/48 blocks.
```

The procedure returns information on all COM files in the directory [VERN].

A.4 SYS.COM

```
$ !
$ ! Displays information about owner, group, or system processes.
$ !
$ ! Turn off verification and save current settings
$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(0)
$ CONTEXT = ""                      ! Initialize PID search context     ❶


$ !
$ ! Output header line.
$ !
$ WRITE SYS$OUTPUT "   PID    Username    Term    Process " + -          ❷
        "name   State   Pri Image"
```

```
$ !
$ ! Output process information.
$ !
$LOOP:
$ !
$ ! Get next PID.  If null, then done.
$ !
$       PID = F$PID(CONTEXT)                    ❸
$       IF PID .EQS. "" THEN GOTO DONE


$ !
$ ! Get image file specification and extract the file name.
$ !
$       IMAGNAME = F$GETJPI(PID,"IMAGNAME")                ❹
$       IMAGNAME = F$PARSE(IMAGNAME,,,"NAME","SYNTAX_ONLY")


$ !
$ ! Get terminal name.  If none, then describe type of process.
$ !
$       TERMINAL = F$GETJPI(PID,"TERMINAL")          ❺
$       IF TERMINAL .EQS. "" THEN -
                TERMINAL = "-"+F$EXTRACT(0,3,F$GETJPI(PID,"MODE"))+"-"
$       IF TERMINAL .EQS. "-INT-" THEN TERMINAL = "-DET-"
$       IF F$GETJPI(PID,"OWNER") .NE. 0 THEN TERMINAL = "-SUB-"


$ !
$ ! Get some more information, put process line together,
$ ! and output it.
$ !
$       LINE = F$FAO("!AS !12AS !7AS !15AS !5AS !2UL/!UL !10AS", -       ❻
                PID,F$GETJPI(PID,"USERNAME"),TERMINAL,-
                F$GETJPI(PID,"PRCNAM"),-
                F$GETJPI(PID,"STATE"),F$GETJPI(PID,"PRI"),-
                F$GETJPI(PID,"PRIB"),IMAGNAME)
$       WRITE SYS$OUTPUT LINE
$       GOTO LOOP


$ !
$ ! Restore verification and exit.
$ !
$DONE:
$  SAVE_VERIFY_PROCEDURE = F$VERIFY(SAVE_VERIFY_PROCEDURE,SAVE_VERIFY_IMAGE)
$  EXIT
```

Notes

❶ The symbol CONTEXT is initialized with a null value. This symbol will be used with the F$PID function to obtain a list of process identification numbers.

❷ The procedure writes a header for the display.

❸ The procedure gets the first process identification (PID) number. If the current process lacks GROUP or WORLD privilege, the PID of the current process is returned. If the current process has GROUP privilege, the first PID in the group list is returned. The first PID in the system list is returned if the current process has WORLD privilege. The function continues to return the next PID in sequence until the last PID is returned. At this point, a null string is returned, and the procedure branches to the end.

❹ The procedure uses the F$GETJPI lexical function to get the image file specification for each PID. The F$PARSE function extracts the file name from the specification returned by the F$GETJPI function.

❺ The procedure uses the F$GETJPI function to get the terminal name for each PID. The F$EXTRACT function extracts the first three characters of the MODE specification returned by F$GETJPI(PID,"MODE") to determine the type of process. The F$GETJPI function is used again to determine whether the process is a subprocess.

❻ The procedure uses the F$GETJPI lexical function to get the username, process name, process state, process priority, and process base priority for each PID returned. The F$FAO lexical function formats this information into a screen display.

Sample Execution

```
$ SYS
   PID     Username    Term    Process name   State   Pri Image
00050011 NETNONPRIV   -NET-   MAIL_14411      LEF    9/4 MAIL
00040013 STOVE        RTA6:   STOVE           LEF    9/4
00140015 MAROT        -DET-   DMFBOACP        HIB    9/8 F11BACP
00080016 THOMPSON     -DET-   MTAOACP         HIB   12/8 MTAAACP
00070017 JUHLES       TTF1:   JUHLES          LEF    9/4
    .
    .
    .
00040018 MARCO        TTA2:   MARCO           HIB    9/4 RTPAD
```

```
0018001A VERN          RTA3:   VERN            LEF    9/4
0033001B YISHA         RTA7:   YISHA           CUR    4/4
0002004A SYSTEM        -DET-   ERRFMT          HIB   12/7 ERRFMT
```

This procedure returns information on all processes on the system. The current process has WORLD privilege.

A.5 GETPARMS.COM

```
$ ! Procedure to count the number of parameters passed to a command
$ ! procedure.  This number is returned as the global symbol PARMCOUNT.


$ !
$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")      ❶
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(0)


$ !
$ IF P1 .EQS. "?" THEN GOTO TELL      ❷
$ !
$ ! Loop to count the number of parameters passed.  Null parameters are
$ ! counted until the last non-null parameter is passed.
$ !


$       COUNT = 0           ❸
$       LASTNONNULL = 0
$ LOOP:
$       IF COUNT .EQ. 8 THEN GOTO END_COUNT
$       COUNT = COUNT + 1
$       IF P'COUNT' .NES. "" THEN LASTNONNULL = COUNT
$ GOTO LOOP
$ !
$ END_COUNT:      ❹


$ !
$ ! Place the number of non-null parameters passed into PARMCOUNT.
$ !
$ PARMCOUNT == LASTNONNULL
$ !
$ ! Restore verification setting, if it was on, before exiting
$ !
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(SAVE_VERIFY_PROCEDURE,SAVE_VERIFY_IMAGE)    ❺
$ EXIT
```

```
$ !
$ TELL:          ❻
$ TYPE SYS$INPUT
       This procedure counts the number of parameters passed to
       another procedure.  This procedure can be called by entering
       the string:
            @GETPARMS 'P1 'P2 'P3 'P4 'P5 'P6 'P7 'P8
       in any procedure.  On return, the global symbol PARMCOUNT
       contains the number of parameters passed to the procedure.
$ !
$ EXIT
```

Notes

❶ The procedure saves the current image and procedure verification settings before setting verification off.

❷ If a question mark character was passed to the procedure as a parameter, the procedure branches to the label TELL (Note 6).

❸ A loop is established to count the number of parameters that were passed to the procedure. The counters COUNT and LASTNONNULL are initialized to 0 before entering the loop. Within the loop, COUNT is incremented and tested against the value 8. If COUNT is equal to 8, the maximum number of parameters has been entered. Each time a non-null parameter is passed, LASTNONNULL is equated to that parameter's number.

Each time the IF command executes, the symbol COUNT has a different value. The first time, the value of COUNT is 1 and the IF command checks P1. The second time, it checks P2, and so on.

❹ When the parameter count reaches 8, the procedure branches to END_COUNT. The symbol LASTNONNULL contains the count of the last non-null parameter passed. This value is placed in the global symbol PARMCOUNT. PARMCOUNT must be defined as a global symbol so that its value can be tested at the calling command level.

❺ The original verification settings are restored. Although you save the current procedure verification setting in the symbol SAVE_VERIFY_PROCEDURE, you no longer need this symbol.

❻ At the label TELL, the TYPE command displays data that is included in the input stream. (In command procedures, the input stream is the command procedure file.) The TYPE command displays instructions on how to use GETPARMS.COM.

Sample Execution

The procedure SORTFILES.COM requires the user to pass three non-null parameters. The SORTFILES.COM procedure can contain the lines:

```
$ @GETPARMS 'P1' 'P2' 'P3' 'P4' 'P5' 'P6' 'P7' 'P8'
$ IF PARMCOUNT .NE. 3 THEN GOTO NOT_ENOUGH
    .
    .
    .
$NOT_ENOUGH:
$ WRITE SYS$OUTPUT -
"Three non-null parameters required.  Type SORTFILES HELP for info."
$ EXIT
```

The procedure SORTFILES.COM can be invoked as follows:

```
$ @SORTFILES DEF 4
Three non-null parameters required. Type SORTFILE HELP for info.
```

In the above example, the procedure SORTFILES.COM defines the symbol GETPARMS as a synonym for @GETPARMS and its parameters. For this procedure to be properly invoked, that is, for the parameters that are passed to SORTFILES to be passed to GETPARMS intact for processing, the synonym must be preceded with an apostrophe.

If the return value from GETPARMS is not 3, SORTFILES issues an error message and exits.

A.6 EDITALL.COM

```
$ ! Procedure to edit files with a specified file type.
$ ! Use P1 to indicate the file type.
$ !
$ ON CONTROL_Y THEN GOTO DONE           ! CTRL/Y action     ❶
$ ON ERROR THEN GOTO DONE
```

```
$ !
$ ! Check for file type parameter.  If one was entered, continue;
$ ! otherwise, prompt for a parameter.
$ !
$ IF P1 .NES. "" THEN GOTO OKAY      ❷
$ INQUIRE P1 "Enter file type of files to edit"


$ !
$ ! List all files with the specified file type and write the DIRECTORY
$ ! output to a file named DIRECT.OUT
$ !


$ OKAY:
$ DIRECTORY/VERSIONS=1/COLUMNS=1 -       ❸
        /NODATE/NOSIZE -
        /NOHEADING/NOTRAILING -
        /OUTPUT=DIRECT.OUT *.'P1'
$ IF .NOT. $STATUS THEN GOTO ERROR_SEC      ❹


$ !
$ OPEN/READ/ERROR=ERROR_SEC DIRFILE DIRECT.OUT      ❺
$ !
$ ! Loop to read directory file
$ !


$ NEWLINE:              ❻
$       READ/END=DONE DIRFILE NAME
$       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:   ! Redefine SYS$INPUT
$       EDIT 'NAME'                               ! Edit the file
$       GOTO NEWLINE
$ !
$ DONE:       ❼
$       CLOSE DIRFILE/ERROR=NOTOPEN               ! Close the file
$ NOTOPEN:
$       DELETE DIRECT.OUT;*                       ! Delete temp file
$ EXIT
$ !
$ ERROR_SEC:
$       WRITE SYS$OUTPUT "Error:  ",F$MESSAGE($STATUS)
$       DELETE DIRECT.OUT;*
$ EXIT
```

Notes

❶ ON commands establish condition handling for this procedure. If any error occurs or if CTRL/Y is pressed at any time during the execution of this procedure, the procedure will branch to the label DONE. Similarly, if any error or severe error occurs, the procedure will branch to the label DONE.

❷ The procedure checks whether a parameter was entered. If not, it will prompt for a file type.

❸ The DIRECTORY command lists all files with the file type specified as P1. The command output is written to the file DIRECT.OUT. The /VERSIONS=1 qualifier requests that only the highest numbered version of each file be listed. The /NOHEADING and /NOTRAILING qualifiers request that no heading lines or directory summaries be included in the output. The /COLUMNS=1 qualifier ensures that one file name per record is given.

❹ The IF command checks the return value from the DIRECTORY command by testing the value of $STATUS. If the DIRECTORY command does not complete successfully, then $STATUS has an even integer value, and the procedure exits.

❺ The OPEN command opens the directory output file and gives it a logical name of DIRFILE.

❻ The READ command reads a line from the DIRECTORY command output into the symbol name NAME. After it reads each line, it redefines the input stream for the edit session with the ASSIGN command. Then, it invokes the editor specifying the symbol NAME as the file specification. When the edit session is completed, the command interpreter reads the next line in the file.

❼ The label DONE is the target label for the /END qualifier on the READ command and the target label for the ON CONTROL_Y and ON ERROR conditions set at the beginning of the procedure. At this label, the procedure performs the necessary cleanup operations.

The CLOSE command closes the DIRECTORY command output file; the /ERROR qualifier specifies the label on the next line in the file. This use of /ERROR will suppress any error message that would be displayed if the directory file is not open. For example, this would occur if CTRL/Y were pressed before the directory file were opened.

The second step in cleanup is to delete the temporary directory file.

Sample Execution

```
$ @EDITALL DAT
*  .
   .
   .
```

The procedure EDITALL is invoked with P1 specified as DAT. The procedure creates a directory listing of all files in the default directory whose file types are DAT and invokes the editor to edit each one.

A.7 FORTUSER.COM

```
$ ! Procedure to create, compile, link, execute, and debug
$ ! FORTRAN programs.  Users can enter only the commands listed
$ ! in the symbol OPTION_TABLE.
$ SET NOCONTROL=Y                                                    ❶
$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(0)
$ OPTION_TABLE = "EDIT/COMPILE/LINK/RUN/EXECUTE/DEBUG/PRINT/HELP/FILE/DONE/"    ❷


$ TYPE SYS$INPUT                                                     ❸
        VAX/VMS FORTRAN Command Interpreter
        Enter name of file with which you would like to work.


$ !
$ ! Set up for initial prompt
$ !
$ PROMPT = "INIT"                                                    ❹
$ GOTO HELP                          ! Print the initial help message
```

```
$ !
$ ! after the first prompting message, use the prompt: Command
$ !
$ INIT:
$ PROMPT = "GET_COMMAND"
$ GOTO FILE                        ! Get initial file name


$ !
$ ! Main command parsing routine.  The routine compares the current
$ ! command against the options in the option table.  When it finds
$ ! a match, it branches to the appropriate label.


$ !
$ GET_COMMAND:
$       ON CONTROL_Y THEN GOTO GET_COMMAND      ! CTRL/Y resets prompt     ❺
$       SET CONTROL=Y
$       ON WARNING THEN GOTO GET_COMMAND        ! If any, reset prompt
$       INQUIRE COMMAND "Command"
$       IF COMMAND .EQS. "" THEN GOTO GET_COMMAND
$       IF F$LOCATE(COMMAND + "/", OPTION_TABLE) .EQ. F$LENGTH(OPTION_TABLE) -   ❻
              THEN GOTO INVALID_COMMAND
$       GOTO 'COMMAND'


$ !
$ INVALID_COMMAND:     ❼
$       WRITE SYS$OUTPUT " Invalid command"
$ !


$ HELP:     ❽
$       TYPE SYS$INPUT
        The commands you can enter are:
        FILE      Name of FORTRAN program in your current
                  default directory.  Subsequent commands
                  process this file.
        EDIT      Edit the program.
        COMPILE   Compile the program with VAX FORTRAN.
        LINK      Link the program to produce an executable image.
        RUN       Run the program's executable image.
        EXECUTE   Same function as COMPILE, LINK, and RUN.
        DEBUG     Run the program under control of the debugger.
        PRINT     Queue the most recent listing file for printing.
        DONE      Return to interactive command level.
        HELP      Print this help message.
        Enter CTRL/Y to restart this session
```

```
$ GOTO 'PROMPT'                                    ❾
$ EDIT:                                            ❿
$       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:
$       EDIT 'FILE_NAME'.FOR
$       GOTO GET_COMMAND
$ COMPILE:
$       FORTRAN 'FILE_NAME'/LIST/OBJECT/DEBUG
$       GOTO GET_COMMAND
$ LINK:
$       LINK 'FILE_NAME'/DEBUG
$       PURGE 'FILE_NAME'.*/KEEP=2
$       GOTO GET_COMMAND
$ RUN:
$       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:
$       RUN/NODEBUG 'FILE_NAME'
$       GOTO GET_COMMAND
$ DEBUG:
$       DEFINE/USER_MODE SYS$INPUT SYS$COMMAND:
$       RUN 'FILE_NAME'
$       GOTO GET_COMMAND
$ EXECUTE:
$       FORTRAN 'FILE_NAME'/LIST/OBJECT
$       LINK/DEBUG 'FILE_NAME'
$       PURGE 'FILE_NAME'.*/KEEP=2
$       RUN/NODEBUG 'FILE_NAME'
$       GOTO GET_COMMAND


$ PRINT:
$       PRINT 'FILE_NAME'
$       GOTO GET_COMMAND
$ BADFILE:                                         ⓫
$       WRITE SYS$OUTPUT "File must be in current default directory."
$ FILE:
$       INQUIRE FILE_NAME "File name"
$       IF FILE_NAME .EQS. "" THEN GOTO FILE
$       IF F$PARSE(FILE_NAME,,,"DIRECTORY") .NES. F$DIRECTORY() -      ⓬
        THEN GOTO BADFILE
$       FILE_NAME = F$PARSE(FILE_NAME,,,"NAME")
$       GOTO GET_COMMAND
$ DONE:
$ EXIT
```

Notes

❶ The SET NOCONTROL=Y command ensures that the user who logs in under the control of this procedure cannot interrupt the procedure or any command or program in it.

❷ The option table lists the commands that the user will be allowed to execute. Each command is separated by a slash.

❸ The procedure introduces itself.

❹ The symbol name PROMPT is given the value of a label in the procedure. When the procedure is initially invoked, this symbol has the value INIT. The HELP command text terminates with a GOTO command that specifies the label PROMPT. When this text is displayed for the first time, the GOTO command results in a branch to the label HELP. This displays the HELP message that explains the commands that you can enter. Then, the procedure branches back to the label INIT, where the value for PROMPT is changed to "GET_COMMAND." Finally, the procedure branches to the label FILE to get a file name. Thereafter, when the help text is displayed, the procedure branches to the label GET_COMMAND to get the next command.

❺ The CTRL/Y condition action is set to return to the label GET_COMMAND, as is the warning condition action. The procedure prompts for a command and continues to prompt, even if nothing is entered. To terminate the session, the command DONE must be used.

❻ The procedure uses the F$LOCATE and F$LENGTH lexical functions to determine whether command is included in the list of options. The F$LOCATE function searches for the user-entered command, followed by a slash. (For example, if you enter EDIT, the procedure searches for EDIT/.) If the command is not included in the option list, then the procedure branches to the label INVALID_COMMAND. If the command is valid, the procedure branches to the appropriate label.

❼ At the label INVALID_COMMAND, the procedure writes an error message and displays the help text that lists the commands that are valid.

❽ The help text lists the commands that are valid. This text is displayed initially. It is also displayed whenever the user issues the HELP command or any invalid command.

❾ At the conclusion of the HELP text, the GOTO command specifies the symbol name PROMPT. When this procedure is first invoked, the symbol has the value INIT. Thereafter, it has the value GET_COMMAND.

⑩ Each valid command in the list has a corresponding entry in the option table and a corresponding label in the command procedure. For the commands that read input from the terminal, for example, EDIT, the procedure contains a DEFINE command that defines the input stream as SYS$COMMAND.

⑪ At the label BADFILE, the procedure displays a message indicating that the file must be in the current directory. Then the procedure prompts for another file name.

⑫ After obtaining a file name, the procedure checks that you have not specified a directory that is different from your current default directory. The procedure then uses the F$PARSE function to extract the file name. (Each command will supply the appropriate default file type.) Next, the procedure branches back to the label GET_COMMAND to get a command to process the file.

Sample Execution

This example illustrates how to use this command procedure as a captive command procedure.

```
Username: CLASS30
Password:
            VAX/VMS Version 4.0
    VAX/VMS FORTRAN Command Interpreter
    Enter name of file with which you would like to work.
    The commands you can enter are:

    FILE        Name of FORTRAN program in your current
                default directory.  Subsequent commands
                process this file.
    EDIT        Edit the program.
    COMPILE     Compile the program with VAX FORTRAN.
    LINK        Link the program to produce an executable image.
    RUN         Run the program's executable image.
    EXECUTE     Same function as COMPILE, LINK and RUN.
    DEBUG       Run the program under control of the debugger.
    PRINT       Queue the most recent listing file for printing.
    DONE        Return to interactive command level.
    HELP        Print this help message.

    Enter CTRL/Y to restart this session
File name: AVERAGE
Command: COMPILE
Command: LINK
Command: RUN
Command: FILE
File name: READFILE
Command: EDIT
```

This sample execution illustrates logging in, the message of the help text being displayed, and some sample commands. First, the user specifies the file AVERAGE, compiles, links, and runs it. Then the user issues the FILE command to begin working on another file.

A.8 LISTER.COM

```
$ ! Procedure to accumulate programmer names and document
$ ! file names.  After all names and files are entered, they are
$ ! sorted in alphabetic order by programmer name.


$ !
$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")      ❶
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(0)
$ !
$ OPEN/WRITE OUTFILE DATA.TMP          ! Create output file     ❷
$ !


$ LOOP:        ❸
$        INQUIRE NAME "Programmer (press RET to quit)"
$        IF NAME .EQS. "" THEN GOTO FINISHED
$        INQUIRE FILE "Document file name"
$        RECORD[0,20]:='NAME'      ❹
$        RECORD[21,20]:='FILE'
$        WRITE OUTFILE RECORD
$        GOTO LOOP
$ FINISHED:
$        CLOSE OUTFILE


$ !
$ DEFINE/USER_MODE SYS$OUTPUT: NL:    ! Suppress sort output
$ SORT/KEY=(POSITION:1,SIZE=20) DATA.TMP DOC.SRT      ❺
$ !
$ OPEN/WRITE OUTFILE DOCUMENT.DAT       ❻
$ WRITE OUTFILE "Programmer Files as of ",F$TIME()
$ WRITE OUTFILE ""
$ RECORD[0,20]:="Programmer Name"
$ RECORD[21,20]:="File Name"
$ WRITE OUTFILE RECORD
$ WRITE OUTFILE ""


$ !
$ CLOSE OUTFILE      ❼
$ APPEND DOC.SRT DOCUMENT.DAT
$ PRINT DOCUMENT.DAT
```

```
$ !
$ INQUIRE CLEAN_UP "Delete temporary files?"      ❽
$ IF CLEAN_UP THEN DELETE DATA.TMP;*,DOC.SRT;*
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(SAVE_VERIFY_PROCEDURE,SAVE_VERIFY_IMAGE)
$ EXIT
```

Notes

❶ LISTER.COM saves the current verification setting and sets verification off.

❷ The OPEN command opens a temporary file for writing.

❸ INQUIRE commands prompt for a programmer name and for a file name. If a null line, signaled by RETURN, is entered in response to the INQUIRE command prompt, the procedure will assume that no more data is to be entered and will branch to the label FINISHED.

❹ After assigning values to the symbols NAME and FILE, the procedure uses the character string overlay format of an assignment statement to construct a value for the symbol RECORD. In columns 1 through 21 of RECORD, the current value of NAME is written. The command interpreter pads the value of NAME with spaces to fill the 20-character length specified.

Similarly, the next 20 columns of RECORD are filled with the value of FILE. Then, the value of RECORD is written to the output file.

❺ After the file has been closed, the procedure sorts the output file DATA.TMP. The DEFINE command directs the SORT command output to the file NL:. Otherwise, these statistics would be displayed on the terminal.

The sort is performed on the first 20 columns, that is, by programmer name.

The sorted output file has the name DOC.SRT.

❻ The procedure creates the final output file, DOCUMENT.DAT, with the OPEN command. The first lines written to the file are header lines, giving a title, the date and time of day, and headings for the columns.

❼ The procedure closes the file DOCUMENT.DAT and appends the sorted output file, DOC.SRT, to it. Then, the output file is queued to the system printer.

❽ Last, the procedure prompts to determine whether to delete the intermediate files. If a true response (T, t, Y, or y) is entered to the INQUIRE command prompt, the files DATA.TMP and DOC.SRT will be deleted. Otherwise, they will be retained.

Sample Execution

```
$ @LISTER
Programmer: WATERS
Document file name: CRYSTAL.CAV
Programmer: JENKINS
Document file name: MARIGOLD.DAT
Programmer: MASON
Document file name: SYSTEM.SRC
Programmer: ANDERSON
Document file name: JUNK.J
Programmer: RET
Delete temporary files:y
```

The output file resulting from this execution of the procedure is:

```
Programmer Files as of 15-APR-1984 16:18:58.79
Programmer Name      File Name
ANDERSON             JUNK.J
JENKINS              MARIGOLD.DAT
MASON                SYSTEM.SRC
WATERS               CRYSTAL.CAV
```

A.9 CALC.COM

```
$ ! Procedure to calculate expressions.  If you enter an
$ ! assignment statement, then CALC.COM evaluates the expression
$ ! and assigns the result to the symbol you specify.  In the next
$ ! iteration, you can use either your symbol or the symbol Q to
$ ! represent the current result.


$ !
$ ! If you enter an expression, then CALC.COM evaluates the
$ ! expression and assigns the result to the symbol Q.  In
$ ! the next iteration, you can use the symbol Q to represent
$ ! the current result.
```

```
$ !
$ SAVE_VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")          ❶
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(0)
$ START:
$     ON WARNING THEN GOTO START
$     INQUIRE STRING "Calc"          ❷
$     IF STRING .EQS. "" THEN GOTO CLEAN_UP
$     IF F$LOCATE("=",STRING) .EQ. F$LENGTH(STRING) THEN GOTO EXPRESSION


$ !
$ ! Execute if string is in the form symbol = expression
$ STATEMENT:          ❸
$     'STRING' ! Execute assignment statements
$     SYMBOL = F$EXTRACT(0,F$LOCATE("=",STRING)-1,STRING) ! get symbol name
$     Q = 'SYMBOL'          ! Set up q for future iterations
$     LINE = F$FAO("Decimal = !SL       Hex = !-!XL      Octal = !-!OL",Q)
$     WRITE SYS$OUTPUT LINE
$     GOTO START
$ !


$ !
$ ! Execute if string is an expression
$ EXPRESSION:          ❹
$     Q = F$INTEGER('STRING')          ! Can use Q in next iteration
$     LINE = F$FAO("Decimal = !SL       Hex = !-!XL      Octal = !-!OL",Q)
$     WRITE SYS$OUTPUT LINE
$     GOTO START


$ !
$ CLEAN_UP:
$ SAVE_VERIFY_PROCEDURE = F$VERIFY(SAVE_VERIFY_PROCEDURE,SAVE_VERIFY_IMAGE)
$ EXIT
```

Notes

❶ The procedure establishes an error handling condition that restarts the procedure. If a warning or an error of greater severity occurs, the procedure will branch to the beginning where it resets the ON condition.

This technique ensures that the procedure will not exit if the user enters an expression incorrectly.

❷ The INQUIRE command prompts for an arithmetic expression. The procedure accepts expressions in either of the formats:

```
name = expression
expression
```

If no expression is entered, the procedure will assume the end of a CALC session and exit.

If you enter input in the format "name = expression" then the procedure continues executing at the label STATEMENT. Otherwise, the procedure branches to the label EXPRESSION.

❸ The procedure executes the assignment statement and assigns the result of the expression to the symbol. Then the procedure extracts the symbol name, and assigns the value of the symbol to Q. This allows you to use either Q or your symbol during the next iteration of the procedure. Next, the procedure displays the result and then branches back to the label START.

❹ The procedure evaluates the expression and assigns the result to the symbol Q. This allows you to use Q during the next iteration of the procedure. Next, the procedure displays the result and then branches back to the label START.

Sample Execution

```
$ @CALC
Calc: 2 * 30
Decimal = 60          Hex = 0000003C  Octal = 00000000074
Calc: Q + 3
Decimal = 63          Hex = 0000003F  Octal = 00000000077
Calc: TOTAL = Q + 4
Decimal = 67          Hex = 00000043  Octal = 00000000103
Calc: 5 + 7
Decimal = 12     Hex = 0000000C  Octal = 00000000014
Calc: RET
$
```

After each prompt from the procedure, the user enters an arithmetic expression. The procedure displays the results in decimal, hexadecimal, and octal. A null line, signaled by <RET> on a line with no data, concludes the CALC session.

A.10 BATCH.COM

```
$ VERIFY_IMAGE = F$ENVIRONMENT("VERIFY_IMAGE")
$ VERIFY_PROCEDURE = F$VERIFY(0)
```

```
$!
$! Turn off verification and save current settings.
$! (This comment must appear after you turn verification
$! off; otherwise it will appear in the batch job log file.)
$!


$!
$! If this is being executed as a batch job,
$! (from the SUBMIT section below) go to the EXECUTE_BATCH_JOB section
$! Otherwise, get the information you need to prepare to execute the
$! batch job.


$!
$ IF F$MODE() .EQS. "BATCH" THEN GOTO EXECUTE_BATCH_JOB        ❶
$!


$!
$! Prepare to submit a command (or a command procedure) as a batch job.
$! First, determine a mnemonic process name for the batch job.  Use the
$! following rules:


$!
$! 1) If the user is executing a single command, then use the verb name.
$!    Strip off any qualifiers that were included with the command.
$! 2) If the user is executing a command procedure, then use the file name.
$! 3) Otherwise, use BATCH.


$!
$ JOB_NAME = P1                ❷
$ IF JOB_NAME .EQS. "" THEN JOB_NAME = "BATCH"
$ IF F$EXTRACT(0,1,JOB_NAME) .EQS. "@" THEN JOB_NAME = F$EXTRACT(1,999,JOB_NAME)
$ JOB_NAME = F$EXTRACT(0,F$LOCATE("/",JOB_NAME),JOB_NAME)
$ JOB_NAME = F$PARSE(JOB_NAME,,,"NAME","SYNTAX_ONLY")
$ IF JOB_NAME .EQS. "" THEN JOB_NAME = "BATCH"
$!


$!
$! Get the current default device and directory.
$!
$ ORIGDIR = F$ENVIRONMENT("DEFAULT")
$!
```

```
$!
$! Concatenate the parameters to form the command string to be executed.
$! If the user did not enter a command string, the symbol COMMAND will have
$! a null value.
$!
$ COMMAND = P1 + " " + P2 + " " + P3 + " " + P4 + " " + -          ❸
          P5 + " " + P6 + " " + P7 + " " + P8
$!


$!
$! If the user is executing a single command and if both the command and the
$! original directory specification are small enough to be passed as
$! parameters to the SUBMIT command, then submit the batch job now
$!
$ IF (P1 .NES. "") .AND. (F$LENGTH(COMMAND) .LE. 255) .AND. -          ❹
       (F$LENGTH(ORIGDIR) .LE. 255) THEN GOTO SUBMIT
$!


$!
$! If the single command to be executed in the batch job is very large, or
$! if you have to prompt for commands to execute in the batch job, then
$! create a temporary command procedure to hold those commands and get the
$! fully expanded name of the command procedure.
$!
$ CREATE_TEMP_FILE:                                                ❺
$   ON CONTROL_Y THEN GOTO CONTROL_Y_HANDLER                       ❻
$   OPEN/WRITE/ERROR=FILE_OPEN_ERROR TEMPFILE SYS$SCRATCH:'JOB_NAME'.TMP
$   FILESPEC = F$SEARCH("SYS$SCRATCH:" + JOB_NAME + ".TMP")


$!
$!  By default, have the batch job continue if it encounters any errors.
$!
$   WRITE TEMPFILE "$ SET NOON"
$!
$!  Either write the single large command to the file, or prompt for
$!  multiple commands and write them to the file.
$!


$   IF COMMAND .NES. "       " THEN GOTO WRITE_LARGE_COMMAND
$
$   LOOP:
$       READ /END_OF_FILE=CLOSE_FILE /PROMPT="Command: " SYS$COMMAND COMMAND
$       IF COMMAND .EQS. "" THEN GOTO CLOSE_FILE
$       WRITE TEMPFILE "$ ",COMMAND
$       GOTO LOOP
$
$ WRITE_LARGE_COMMAND:
$   WRITE TEMPFILE "$ ",COMMAND
$
```

```
$!
$! Finish the temporary file by defining a symbol so that you will know
$! the name of the command procedure to delete and then close the file.
$! Define the symbol COMMAND to mean "execute the command procedure
$! you have just created".  Then submit the batch job and execute
$! this command procedure in the batch job.
$!

$ CLOSE_FILE:                                    ❼
$   WRITE TEMPFILE "$ BATCH$DELETE_FILESPEC == """,FILESPEC,""""
$   CLOSE TEMPFILE
$   ON CONTROL_Y THEN EXIT
$   COMMAND = "@" + FILESPEC
$!
$!
$! Submit BATCH.COM as a batch job, and pass it two parameters.
$! P1 is the command (or name of the command procedure) to execute.
$! P2 is the directory from which to execute the command.
$!

$ SUBMIT:                                        ❽
$   SUBMIT/NOTIFY/NOPRINT 'F$ENVIRONMENT("PROCEDURE")' /NAME='JOB_NAME' -
        /PARAMETERS=("''COMMAND'","''ORIGDIR'")
$   GOTO EXIT
$!
$!
$! The user pressed a control-y while the temporary
$! command procedure was open.  Close the command procedure,
$! delete it if it exists, and exit.
$!

$ CONTROL_Y_HANDLER:                             ❾
$   CLOSE TEMPFILE
$   IF F$TYPE(FILESPEC) .NES. "" THEN DELETE/NOLOG 'FILESPEC'
$   WRITE SYS$OUTPUT "CTRL/Y caused the command procedure to abort."
$   GOTO EXIT
$!
$!
$! The temporary command procedure could not be created.
$! Notify the user and exit.
$!

$ FILE_OPEN_ERROR:                               ❿
$   WRITE SYS$OUTPUT "Could not create sys$scratch:",job_name,".tmp"
$   WRITE SYS$OUTPUT "Please correct the situation and try again."
$!
$!
$! Restore the verification states and exit.
$!
```

```
$ EXIT:                                    ⓫
$  VERIFY_PROCEDURE = F$VERIFY(VERIFY_PROCEDURE,VERIFY_IMAGE)
$  EXIT
$!
$!
$! BATCH.COM was invoked as a batch job.  P1 contains the command
$! to execute and P2 the default directory specification.
$! Return a status code that indicates the termination status of P1.
$!


$ EXECUTE_BATCH_JOB:                              ⓬
$    SET NOON
$    VERIFY_PROCEDURE = F$VERIFY(VERIFY_PROCEDURE,VERIFY_IMAGE)
$    SET DEFAULT 'P2'
$    'P1'
$    IF F$TYPE(BATCH$DELETE_FILESPEC) .EQS. "" THEN EXIT $STATUS
$    STATUS = $STATUS
$    DELETE /NOLOG 'BATCH$DELETE_FILESPEC'
$    EXIT STATUS
```

Notes

❶ This IF statement tests whether BATCH.COM is executing in batch mode. When you invoke BATCH.COM interactively, you provide (as parameters) a command string or a command procedure that is to be executed as a batch job. If you do not supply any parameters, then BATCH.COM prompts you for commands, writes these commands to a file, and then executes this command procedure as a batch job. After BATCH.COM prepares your command(s) for execution from a batch job, it uses the SUBMIT command to submit itself as a batch job and execute your command(s) from this job. (See Note 8.) When you invoke BATCH.COM as a batch job, the procedure branches to the label EXECUTE_BATCH_JOB.

❷ These commands prepare the batch job for execution. First, the procedure constructs a name for the batch job. If a command string was passed, then BATCH.COM uses the verb name as the job name. If a command procedure was passed, then BATCH.COM uses the file name. If no input was passed, then BATCH.COM names the job BATCH.

❸ The parameters are concatenated to form the command string to be executed. The command string is assigned to the symbol COMMAND.

❹ The SUBMIT command cannot pass a parameter that is greater than 5 characters. Therefore, the procedure tests that the command string and directory name are less than 255 characters long. If both strings are less than 255 characters (and if the user did, in fact, pass a command string) then the procedure branches to the label SUBMIT.

❺ The procedure establishes a CTRL/Y handler so clean-up operations will be performed if the user presses CTRL/Y during this section of the command procedure.

❻ The procedure creates a temporary file to contain the commands to be executed. If the user supplied a long command string, the procedure branches to WRITE_LARGE_COMMAND and writes this command to the temporary file. Otherwise, the procedure prompts for the commands to be executed. Each command is written to the temporary file.

❼ Before you close the temporary file, write a symbol assignment statement to the file. This statement assigns the file name for the temporary file to the symbol BATCH$DELETE_FILESPEC. After closing the temporary file, the procedure resets the default CTRL/Y handler. Then the procedure defines the symbol COMMAND so that, when executed, COMMAND will invoke the temporary command file.

❽ The procedure submits itself as a batch job, using the defined job name. (See Note 2.) The procedure also passes two parameters: the command or command procedure to be executed, and the directory from which the command should be executed. Then the procedure branches to the label EXIT. (See Note 11.)

❾ This section performs clean-up operations if the user types CTRL/Y while the temporary file is being created.

❿ This section writes an error message if the temporary file cannot be created.

⓫ The procedure resets the original verification settings and then exits.

⓬ These commands are executed when BATCH.COM runs in batch mode. First, ON error handling is disabled and the user's default verification settings are set. Then the default is set to the directory indicated by P2, and the command indicated by P1 is executed. If a temporary file was created, this file is deleted. The completion status for P1 is saved before deleting BATCH$DELETE_FILESPEC. This completion status is returned by the EXIT command.

Sample Execution

```
$ @BATCH RUN MYPROG
Job RUN (queue SYS$BATCH, entry 1715) started on SYS$BATCH
```

This example uses BATCH.COM to run a program from within a batch job.

B Submitting Batch Jobs Through the Card Reader

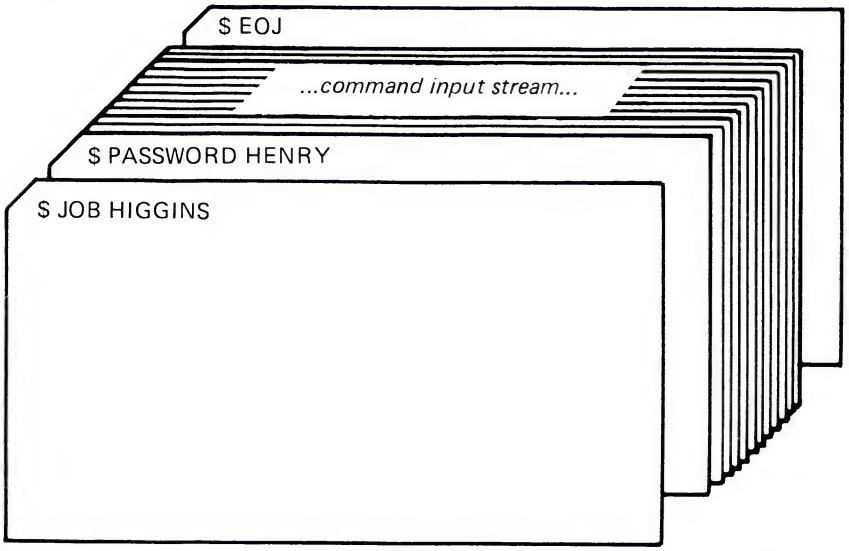
When you submit a batch job through a system card reader, you must precede the card deck containing the command procedure with cards containing JOB and PASSWORD commands. These cards specify your user name and password and, when executed, effect a login for you. The last card in the deck must contain the EOJ (End of Job) command. The EOJ card is equivalent to logging out. You can also use an overpunch card instead of an EOJ card to signal the end of a job. To do this, use an EOF card (12-11-0-1-6-7-8-9) overpunch or use the EOJ command.

Figure B–1 illustrates a card reader batch job.

When the system reads a job from the card reader, it validates the user name and password specified on the JOB and PASSWORD cards. Then, it copies the entire card deck into a temporary disk file named INPBATCH.COM in your default disk and directory and queues the job for batch execution. Thereafter, processing is the same as for jobs submitted interactively with the SUBMIT command. When the batch job is completed, the operating system deletes the INPBATCH.COM file.

Note that you can prevent other users from seeing your password by suppressing printing when you keypunch the PASSWORD card.

Figure B–1 A Card Reader Batch Job



B.1 Translation Modes

The system can read cards that were punched on an 026 punch or an 029 punch. By default, the translation mode is 029; that is, the system reads cards from an 029 punch. However, you can change the translation mode by using:

- The SET CARD_READER command
- Translation mode cards

Use the SET CARD_READER command with the /026 or /029 qualifier to set the card reader to accept cards from either an 026 or an 029 card punch. Create translation mode cards by overpunching cards; an 026 punch is indicated by an 026 translation mode card (12-2-4-8 overpunch). An 029 card punch is indicated by an 029 translation mode card (12-0-2-4-6-8 overpunch).

B.2 Passing Data to Commands and Images

To pass data to commands and images in batch jobs that you submit through a card reader, you can:

- Include the data in the command procedure by placing the data on the lines after the command or image that uses the data. Use the DECK and EOD commands if the data lines begin with dollar signs.
- Temporarily redefine SYS$INPUT as a file by using the DEFINE/USER_MODE command.

C Summary of Lexical Functions

The following list summarizes each lexical function and its arguments. For complete information on each function, including examples, see the *VAX/VMS DCL Dictionary*.

F$CVSI(bit-position ,count ,integer)
Extracts bit fields from character string data and converts the result, as a signed value, to an integer.

F$CVTIME(time)
Retrieves information about an absolute, combination, or delta time string.

F$CVUI(bit-position ,count ,integer)
Extracts bit fields from character string data and converts the result, as an unsigned value, to an integer.

F$DIRECTORY()
Returns the current default directory name string.

F$EDIT(string ,edit-list)
Edits a character string based on the edits specified.

F$ELEMENT(element-number ,delimiter ,string)
Extracts an element from a string in which the elements are separated by a specified delimiter.

F$ENVIRONMENT(item)
Obtains information about the DCL command environment.

F$EXTRACT(offset ,length ,string)
Extracts a substring from a character string expression.

F$FAO(control-string [,arg1 ,arg2...arg15])
Invokes the $FAO system service to convert the specified control string to a formatted ASCII output string.

F$FILE_ATTRIBUTES(file-spec ,item)
Returns attribute information for a specified file.

F$GETDVI(device-name ,item)
Invokes the $GETDVI system service to return a specified item of information for a specified device.

F$GETJPI(pid ,item)
Invokes the $GETJPI system service to return accounting, status, and identification information for a process.

F$GETSYI(item [,node])
Invokes the $GETSYI system service to return status and identification information about the local system, or about a node in the local cluster, if your system is part of a cluster.

F$IDENTIFIER(identifier,conversion-type)
Converts an identifier in named format to its integer equivalent, or vice versa.

F$INTEGER(expression)
Returns the integer equivalent of the result of the specified expression.

F$LENGTH(string)
Returns the length of a specified string.

F$LOCATE(substring ,string)
Locates a character or character substring within a string and returns its offset within the string.

F$LOGICAL(logical-name)
Translates a logical name and returns the equivalence name string.

F$MESSAGE(status-code)
Returns the message text associated with a specified system status code value.

F$MODE()
Shows the mode in which a process is executing.

F$PARSE(file-spec [,default-spec] [,related-spec] [,field])
Invokes the $PARSE RMS service to parse a file specification and return either the expanded file specification or the particular file specification field that you request.

F$PID(context-symbol)
For each invocation, returns the next process identification number in sequence.

F$PRIVILEGE(priv-states)
Returns a value of "TRUE" or "FALSE" depending on whether your current process privileges match the privileges listed in the argument.

F$PROCESS()
Returns the current process name string.

F$SEARCH(file-spec [,stream-id])
Invokes the $SEARCH RMS service to search a directory file, and returns the full file specification for a file you name.

F$SETPRV(priv-states)
Sets the specified privileges and returns a list of keywords indicating the previous state of these privileges for the current process.

F$STRING(expression)
Returns the string equivalent of the result of the specified expression.

F$TIME()
Returns the current date and time of day, in the format dd-mm-yyy hh:mm:ss.cc.

F$TRNLNM(logical-name [,table] [,index] [,mode] [,case] [,item])
Translates a logical name and returns the equivalence name string or the requested attributes of the logical name.

F$TYPE(symbol-name)
Determines the data type of a symbol.

F$USER()
Returns the current user identification code (UIC).

F$VERIFY([procedure-value] [,image-value])

Returns the integer 1 if command procedure verification is set on; returns the integer 0 if command procedure verification is set off. The F$VERIFY function also can set new verification states.

D Commands Frequently Used in Command Procedures

The following DCL commands are frequently used in command procedures. This list also includes commands that you use to control interactive command procedures and batch jobs.

Name	Function
@	Executes a command procedure.
=	Symbol assignment; equates a local symbol name to an integer expression or character string expression.
==	Symbol assignment; equates a global symbol name to an integer expression or character string expression.
:=	String assignment; equates a local symbol name to a character string. Commonly used to equate a symbol to a command string without having to enclose the string in quotation marks.
:==	String assignment; equates a global symbol name to a character string. Commonly used to equate a symbol to a command string without having to enclose the string in quotation marks.
ASSIGN	Equates a logical name to a physical device name, to a complete file specification, or another logical name, and places the equivalence name string in the process, group, or system logical name table.
CLOSE	Closes a file that was opened for input or output with the OPEN command and deassigns the logical name specified when the file was opened.
CONTINUE	Resumes execution of a command procedure or image that was interrupted by pressing CTRL/Y or CTRL/C.

Name	Function
DEASSIGN	Cancels a logical name assignment made with the ALLOCATE, ASSIGN, or DEFINE command.
DECK	Marks the beginning of an input stream for a command procedure when the first nonblank character in any data record in the input stream is a dollar sign ($).
DEFINE	Equates a logical name to a physical device name, to a complete file specification, or to another logical name, and places the equivalence name string in the process, group, or system logical name table.
DELETE/ENTRY	Deletes one or more entries from a print or batch job queue.
DELETE/SYMBOL	Deletes a symbol definition from a local symbol table or from the global symbol table.
EOD	Marks the end of an input stream for a command procedure.
EOJ	Marks the end of a batch job submitted through a system card reader.
EXIT	Terminates processing of the current command procedure.
GOTO	Transfers control to a labeled statement in a command procedure.
IF...THEN	Tests the value of an expression and executes a command if the test is true.
INQUIRE	Requests interactive assignment of a value for a local or global symbol during the execution of a command procedure.
JOB	Marks the beginning of a batch job submitted through a system card reader.
ON...THEN	Defines the default courses of action when a command or program executed within a command procedure (1) encounters an error condition or (2) is interrupted by CTRL/Y.
OPEN	Opens a file for reading or writing.

Name	Function
PASSWORD	Specifies the password associated with the user name specified on a JOB card for a batch job submitted through a card reader.
PRINT	Queues one or more files for printing, either on a default system printer or a specified device.
READ	Reads a single record from a specified input file and assigns the contents of the record to a specified logical name.
SET CARD_READER	Defines the default ASCII translation mode for a card reader.
SET CONTROL	Enables interrupts caused by CTRL/Y and CTRL/T.
SET NOCONTROL	Disables interrupts caused by CTRL/Y and CTRL/T.
SET NOON	Prevents the command interpreter from performing error checking following the execution of commands.
SET NOVERIFY	Prevents command lines in a command procedure from being displayed at a terminal or printed in a batch job log file.
SET ON	Causes the command interpreter to perform error checking following the execution of commands.
SET OUTPUT_RATE	Sets the rate at which output is written to a batch job log file.
SET RESTART_VALUE	Establishes a test value for restarting portions of a batch job.
SET QUEUE/ENTRY	Changes the current status or attributes of a file that is queued for printing or for batch job execution but not yet processed.
SET VERIFY	Causes command lines in a command procedure to be displayed at a terminal or printed in a batch job log file.
SHOW QUEUE	Displays the current status of entries in the print and/or batch job queues.
STOP/ABORT	Aborts a job that is currently being printed.

Name	Function
STOP/ENTRY	Deletes an entry from a batch queue while it is running.
STOP/REQUEUE	Stops the printing of the job currently being printed and places that job at the end of the output queue.
SUBMIT	Enters one or more command procedures in the batch job queue.
SYNCHRONIZE	Places the process issuing this command into a wait state until a specified batch job completes execution.
WAIT	Places the current process in a wait state until a specified period of time has elapsed.
WRITE	Writes a record to a specified output file.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Did you find errors in this manual? If so, specify the error and the page number.

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line